Spring 3-2019

Malgazer: An Automated Malware Classifier With Running Window Entropy and Machine Learning

Keith Jones
Dakota State University

MALGAZER: AN AUTOMATED MALWARE CLASSIFIER WITH RUNNING WINDOW ENTROPY AND MACHINE LEARNING

A dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Ph.D.

in

Cyber Operations

March 2019

By
Keith J. Jones

Dissertation Committee:

Dr. Yong Wang
Dr. Jun Liu
Dr. Wayne Pauli
Dr. Joshua Pauli
Dr. Joshua Stroschein

DSU
DAKOTA STATE

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Keith Jones

Dissertation Title: Malgazer: An Automated Malware Classifier with Running Window Entropy and Machine Learning

Dissertation Chair/Co-Chair: _____ Date: 03/26/2019

Committee member: _____ Date: 3/26/19

Committee member: _____ Date: 3/26/19

Committee member: _____ Date: 3/27/19

_____ 3/26/19

Original to Office of Graduate Studies and Research
Acid-free copies with written reports to library

ACKNOWLEDGMENT

I am very grateful to my coworkers at Cylance, Inc. for many ideas, daily inspiration, promotion of education, and understanding about the requirements school entails. I would also like to thank Lakshmanan Nataraj and colleagues from University of California, Santa Barbara for personal communication and assistance while implementing the GIST method. I am very grateful to all of my professors at Dakota State University, but most importantly I owe a debt of gratitude to the committee of professors that agreed to be part of this dissertation: Dr. Wayne Pauli, Dr. Joshua Pauli, Dr. Joshua Stroschein, and Dr. Jun Liu. My committee chair Dr. Yong Wang, with whom I took two classes and co-authored a paper over the past three years, will always have my thanks for constantly making himself available, his guidance, his insightful questions, and his invaluable assistance.

This dissertation could never have been written without the support of family and friends. I would like to thank my amazing wife Andrea and my wonderful kids Aiden, Madeleine, and Charlotte for the numerous times I missed dinners, practices, evenings, weekends, and games. I would like to thank my father-in-law Dr. Jan Amsterburg for showing me it is not too late to finish your Ph.D and for being a great role model. I would also like to acknowledge my mother-in-law Carol Amsterburg for her encouragement and grammar assistance. I am also appreciative for the support of Audrey Martini, who has believed in me since 1994. Last but not least, I would like to acknowledge my late grandmother Gertrude “Mema” Lerminiaux, who can finally rest in peace because I completed this last promise to her.

ABSTRACT

This dissertation explores functional malware classification using running window entropy and machine learning classifiers. This topic was under researched in the prior literature, but the implications are important for malware defense. This dissertation will present six new design science artifacts. The first artifact was a generalized machine learning based malware classifier model. This model was used to categorize and explain the gaps in the prior literature. This artifact was also used to compare the prior literature to the classifiers created in this dissertation, herein referred to as "Malgazer" classifiers.

Running window entropy data was required, but the algorithm was too slow to compute at scale. This dissertation presents an optimized version of the algorithm that requires less than 2% of the time of the original algorithm. Next, the classifications for the malware samples were required, but there was no one unified and consistent source for this information. One of the design science artifacts was the method to determine the classifications from publicly available resources.

Once the running window entropy data was computed and the functional classifications were collected, the machine learning algorithms were trained at scale so that one individual could complete over 200 computationally intensive experiments for this dissertation. The method to scale the computations was an instantiation design science artifact. The trained classifiers were another design science artifact. Lastly, a web application was developed so that the classifiers could be utilized by those without a programming background. This was the last design science artifact created by this research.

Once the classifiers were developed, they were compared to prior literature theoretically and empirically. A malware classification method from prior literature was chosen (referred to herein as "GIST") for an empirical comparison to the Malgazer classifiers. The best Malgazer classifier produced an accuracy of approximately 95%, which was around 0.76% more accurate than the GIST method on the same data sets. Then, the Malgazer classifier was compared to the prior literature theoretically, based upon the empirical analysis with GIST, and Malgazer performed at least as well as the prior literature. While the data, methods, and source code are open sourced from this research, most prior literature did not provide enough information or data

to replicate and verify each method. This prevented a full and true comparison to prior literature, but it did not prevent recommending the Malgazer classifier for some use cases.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Keith J. Jones

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND BACKGROUND

Motivation

There were on average 200,000+ new malware threats released into the internet each day in 2017, and that number generally grows each year (Panda Security, 2017). Some commercial malware detection tools claimed 95% efficacy rates for malware detection (Saxe & Berlin, 2015) or beyond (Cylance Inc., 2016). Unfortunately, malware detection and classification are often used interchangeably in prior literature and in industry. Malware detection determines if a sample was malware or not, while malware classification determines what type of behavior, function and family the malware exhibited. Operationally, detection and classification algorithms can be separated to provide critical real-time sensitive information for the detection, and then deliver more time tolerant, richer information to the classifier. In most real-world scenarios, the classification information comes well after the sample was caught and blocked prior to execution by the detector. As detection efficacy continues to improve in practice, classification efficacy is a more complex, interesting, and richer problem that requires more research.

As shown in Figure 1, malware detection is a subset[1] of the overall malware classification problem, and it is the classification problem this dissertation will examine. This dissertation will present an automated machine-learning based method of malware classification that could be applied independently after the detection method, thereby relaxing the time sensitivity requirement that the overall classification of the sample must occur at the same time it was detected and blocked. The additional time between detection and classification would allow for the computation of additional rich data from the sample, such as running window entropy

[1] With detection, there are only two classes in a classification problem. One class is “malware” and the other is “not malware” (or “benign”).

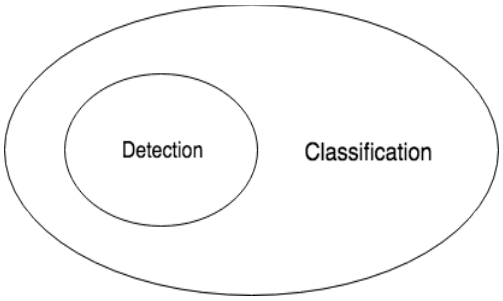


Figure 1. The malware classification and detection problems

(RWE). RWE would normally not be available during the time sensitive detection phase because of the additional time required for more complex computations.

For simplicity, the classifier and design science artifacts developed in this dissertation will be collectively referred to as "Malgazer". Malgazer's source code is open source and can be downloaded at (Jones, 2017b).

Running Window Entropy

As a basis for Malgazer, running window entropy, sometimes applied with slight modifications and called "structural entropy" in prior literature (Baysa, 2013; Lyda & Hamrock, 2007; Sorokin, 2011; Wojnowicz, Chisholm, & Wolff, 2016; Wojnowicz, Chisholm, Wolff, & Zhao, 2016; Wolff & Chisholm, 2015), provides a localized view of entropy data near a particular offset within the file. Running window entropy, a series of values, is different than full file entropy, a scalar value. Full file entropy can be easily manipulated by malware authors by appending a set of chosen bytes to the malware sample. For example, adding many zeros to the end of a malware sample will trend the overall file entropy downward, while it would not change the previous values in the running window entropy series or modify the malware sample's execution behavior.

While the example above applies to the full sample's entropy, the same method could be applied to sections within an executable file and achieve the same effect if the scalar entropy value of a section is calculated. A "section" could be a Windows PE file section, or it could be a section of an ELF file, as most binary executable files use sections. Therefore, simply computing the entropy of certain sections or the whole file may not lead to accurate results because the malware author could have easily manipulated the entropy in this manner.

Entropy, as originally defined by Shannon for information theory (Shannon, 1948), measures the amount of useful information in data. It is important to know that a higher entropy value equates to more relevant information in the data while a lower entropy value equates to less relevant information. A lower entropy value indicates the data was predictable while a high entropy value indicates the data was more random and contained more useful, unpredictable, information. This type of additional data, if applied appropriately, could increase automated classification accuracy. As discussed in later chapters, Malgazer classified running window entropy data with approximately 95% accuracy. 95% accuracy was a slight improvement over the GIST method, a prior malware classification algorithm discussed in the next chapter.

Improved accuracy of automated malware classifiers would allow for reductions of intervening human effort and costs during this compulsory malware analysis stage. This dissertation examines running window entropy as a means for classification in an effort to reduce the costs associated with malware analysis. More information about the use of entropy for malware analysis is presented in the second half of this chapter.

The computation of running window entropy required a computationally expensive double "for" loop that depended on the running window size and the malware sample size. To calculate the running window entropy of 200,000+ new malware variants each day, this algorithm must be implemented efficiently for use in practice. This dissertation will explore one such algorithm, developed by this author, that included optimizations to reduce the time required for this computation. The optimized algorithm will be presented in chapter 4.

The Malware Classification Problem

Before any classifier can be developed, the generalized concepts of machine learning based malware classification must be understood and agreed upon. The literature review for this dissertation did not uncover a consistent, unified, and generalized model for the machine learning based malware classification problem. Articles that attempted, such as (Shabtai, Moskovitch, Elovici, & Glezer, 2009), did not present a model using mathematical concepts that could be useful for future research. In order for different methods to be compared, the individual components must be organized into a generalized model so that the methods can be compared in a one to one fashion at the component level.

In Figure 2, a generalized model of the machine learning based classification system is presented with each component clearly identified in the model. This generalized machine learning classification model will be used for the remainder of this dissertation. The model introduced in Figure 2 is applied to the prior research in chapter 2, thereby demonstrating the validity and need of such model.

The model in Figure 2 will be referred to as the "generalized machine learning classification model" and is not to be confused with the machine learning model inside a specific classifier ("MLM" in Figure 2). The classifier developed in this dissertation will be referred to as "Malgazer" and will be based on running window entropy using the framework of the generalized machine learning classification model. Methods from prior literature will be referenced by its type as appropriate, such as "GIST" (Laks, 2013, 2014; Lakshmanan Nataraj, Karthikeyan, Jacob, & Manjunath, 2011) or the method number. The method number identifications will be made in chapter 2.

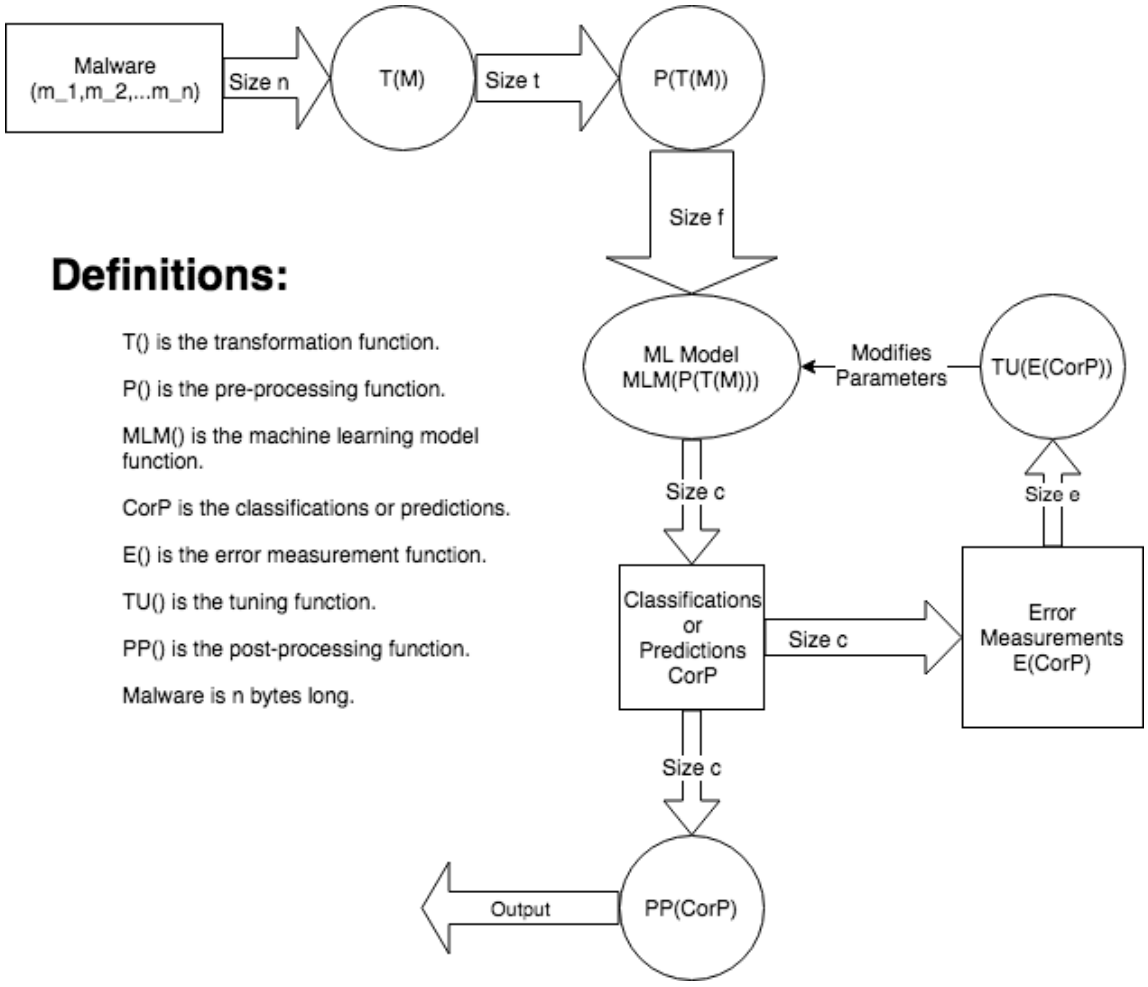


Figure 2. A generalized machine learning based malware classifier

A description of the generalized model in Figure 2 follows. The malware is represented by the vector "M", where the vector is of size "n" to correspond to the number of bytes in the malware sample. This vector is the input to a transformation function "T". It is also assumed file metadata is captured within "T". "T" is a function that computes outputs based upon the malware input. An example of "T" may be the representation of the malware executable in memory after it uncompressed itself. Another example of "T" could be the calculation of the running window entropy series from the malware input. The output of the function "T" is of size "t". The next node is the function "P", which represents pre-processing operations on the data. An example of "P" could be the normalization and standardization of the running window entropy features for the ML model so that the intensive computations are much smaller (Eremenko & de Ponteves, 2017). The output of this node is of size "f", which is the number of

features fed into the machine learning model “MLM”. Machine learning model will be discussed more in depth later in this in chapter.

The machine learning model provides an output of classifications or predictions that is of size “c”. The classifications or predictions (CorP) could be measured in order to provide error measurements using the function “E”, of output size “e”, for the model’s performance. The error measurements could be provided to a tuning function, identified as “TU”, that modifies the machine learning model’s behavior. As an example, the “TU” function could tune parameters of “MLM” based upon the area under the curve (AUC) for the receiver operating characteristic (ROC) curve (Hanley & McNeil, 1982), a measurement that is often used as to determine the accuracy of a classifier.

The classifications or predictions could also be post-processed, identified as the function “PP”. An example of “PP” could be preparation of the classifications as input to another neural network or machine learning algorithm. In all cases, the choice of function dictates the output size from a given input size.

This mathematically based, generalized machine learning based malware classification model will be used throughout this dissertation. Formally, a generalized machine learning based malware classifier is defined as:

Definition 1. A generalized machine learning based malware classifier

A generalized machine learning based malware classifier is constructed from a set C, containing the following functions from Figure 2: T(), P(), MLM(), E(), TU(), and PP().

T() is the transformation function. P() is the pre-processing function. MLM() is the machine learning model. E() is the error measurement function. TU() is the machine learning model tuning function and PP() is the post processing function.

Based upon the conceptual model presented previously, the answer to the problem of automated machine learning malware classification is a search problem for adequate parameters of the set "C" according to a given set of requirements for a specific use case. Note that the requirements could be defined based upon the error measurements in the right most square of Figure 2.

Malgazer will use running window entropy as features for classification, but the length of the running window entropy series of a file will depend on the size of the malware sample. Most machine learning based classifiers use a fixed number of features, so the input vector must be normalized to the same size for any malware sample regardless of the size. This is another example of the preprocessing function "P" in Figure 2. For example, a large running window entropy series from each sample could be normalized, using interpolation and down sampling, to an array of 1,024 values. This is similar to reducing the size of input images when machine learning based classifiers are applied for the purposes of handwriting recognition (Eremenko & de Ponteves, 2017).

Input Data Sets

The efficacy of most machine learning classification algorithms depends heavily upon the training phase. The training phase for supervised machine learning classification algorithms, further discussed in chapter 3, simply requires a set of features that are already classified. During the training phase, the machine learning algorithm ("MLM" in Figure 2 and Definition 1) "learns the best" internal parameters of the models based upon the input training data set and the

already known classifications. This type of machine learning is called supervised learning, since the classifications of the training data sets are already known. Supervised machine learning is different than unsupervised machine learning such that unsupervised learning does not require the training data to be labeled and the unsupervised algorithm decides the classifications for sets of input features on its own. Since this dissertation will focus on the functional classification of malware samples rather than allowing an algorithm to determine clusters of samples, supervised machine learning algorithms are more relevant to this research problem.

Additionally, the training data quality also effects the overall classifier accuracy. The training data is separated into two smaller sets for training purposes: the training data set and a testing data set. A testing data set is used to measure the accuracy of a classifier that was created with the corresponding training data set. Therefore, a testing data set that does not accurately reflect the training data set will drive the classifier's testing accuracy downward. On the other hand, a testing data set that matches the training data set too closely could artificially inflate the classifier's testing accuracy and cause the accuracy in real world applications to be lower than expected. Therefore, in addition to Figure 2 and Definition 1, the input data sets must be declared in order for validation and reproducibility of any proposed classifier. The input data sets will be discussed for prior research in chapter 2. The input data set used to develop Malgazer will be discussed further in chapter 3.

Classifier Training And Implementation

Malgazer was implemented using the concepts presented in Figure 3. The large upper box encapsulates the training and testing phases. The training phase uses the training data set and the testing phase uses the testing data set as discussed previously[2]. The training phase forces the classifier to learn the internal "best" parameters for the data set while the testing phase measures the accuracy of the classifier on new samples for which the user already knows the correct classifications. In the lower half of Figure 3, the prediction phase box represents the implementation of the classifier on unknown malware samples. The prediction phase occurs when the Malgazer classifier is in production.

[2] Technically, the training data set is also tested for accuracy during the training phase, but it can be optionally tested again in the testing phase. Doing so would lead to the same results for deterministic classifiers.

Statement Of The Research Problem

The problem investigated in this dissertation is to develop an automated machine learning based classifier (Malgazer) using running window entropy as the feature space that will classify new malware samples functionally. This problem will be solved using a design science research methodology (Arnould, Hevner, March, & Park, 2004). The classifier will be built with the generalized machine learning malware classifier outlined in Figure 2 and Definition 1 by selecting the appropriate parameters in the classifier set "C" and using running window entropy as the input feature space. The accuracy of the classifier will be measured by the area under the curve (AUC) of the receiver operating characteristic (ROC) and using a confusion matrix (CM) to compute the classification accuracy. Using these measurements, and other appropriate statistical measurements as needed, the parameters of set "C" will be determined using an organized searching methodology. The efficacy of Malgazer will then be compared to prior research theoretically and empirically. The detailed methodology for the proposed solution to this problem will be presented in chapter 3.

The Hypothesis

The null hypothesis for this research problem is if the Malgazer classifier cannot be constructed from the running window entropy feature space. In this case, all instances of trained Malgazer classifiers would perform worse than random guessing. The hypothesis in this dissertation that a Malgazer classifier can be developed will be proven by the existence of a classifier that can perform better than random guessing. It is the existence of such a classifier that this dissertation presents using a programmatic searching methodology.

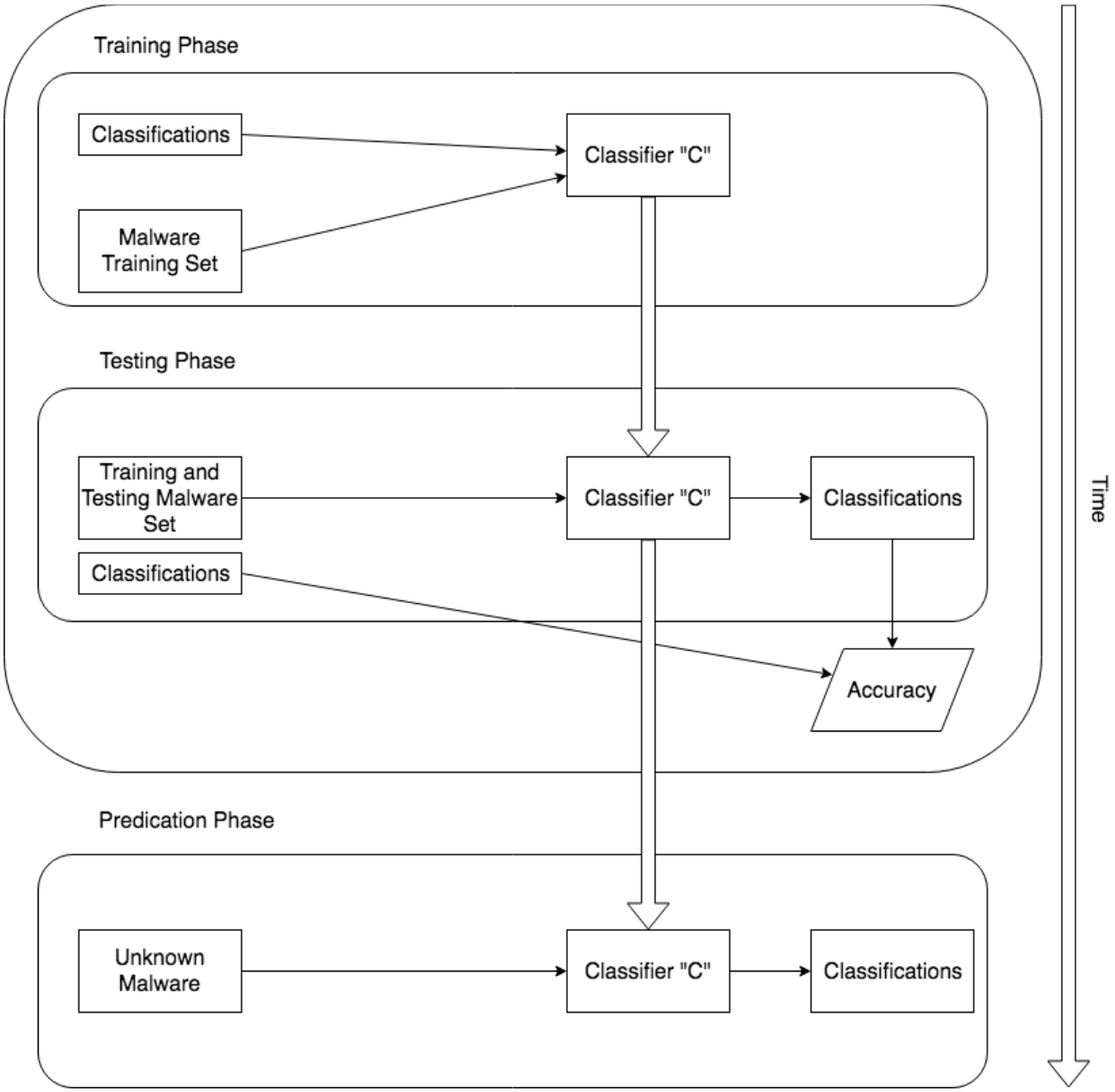


Figure 3. The classifier implementation

Comparison To Prior Methods

The classifier developed in this dissertation will be compared to the reported accuracies of classifiers from prior literature in chapter 7. However, it is important to keep in mind that the reported accuracies from prior literature are highly dependent on the input training malware data set, among other numerous factors, that are difficult to discern, source and replicate from such literature alone. To remove the variable of input data training sets as a factor, the Malgazer classifier developed in this dissertation will be closely compared empirically to a malware

classifier using GIST features (Laks, 2013, 2014; Lakshmanan Nataraj et al., 2011). Contact with an author of the GIST paper, for which this author is very grateful, allowed for replication of their methods using the Malgazer training data. Authors of other similar works were contacted as well, but the GIST authors provided the details required to fully implement their methodology while authors of other methods did not reply or replied well after the GIST methodology was implemented for this research.

The empirical comparison will allow for the GIST method to be directly compared to Malgazer using the same training and testing malware samples. This comparison will remove the input data set variable between the GIST method and the Malgazer classifier. Since the input set of malware samples influences the accuracy of a machine learning based classifier, this comparison will more accurately reflect the efficacy of Malgazer versus the GIST malware classifier rather than a simple, and very problematic, comparison of best accuracies from different training data sets.

Objectives and Contributions

The research problem will be solved through several newly developed design science artifacts (Arnould et al., 2004). Each artifact and the type of artifact will be discussed in its own sub-section that follows.

A Generalized Machine Learning Based Malware Classifier Model

The first design science artifact is the model presented in Figure 2 and Definition 1. This model will be applied to prior research in chapter 2, for each publication reviewed. The definition of this model for each publication facilitated the one to one component comparisons to Malgazer. By using this generalized classification model, the strengths, weaknesses, and differences of each method's component can be categorized and compared. By associating prior literature with the generalized classification model for comparison purposes, the usefulness of this design science artifact will become apparent. This model is a design science model artifact.

An Optimized Running Window Entropy Algorithm

As mentioned previously, the computation of running window entropy can be minimized by optimizing the algorithm with respect to time. In chapter 4, an algorithm developed by this author minimizing the computation time of running window entropy is presented. This algorithm is a design science method artifact. This algorithm was published at a peer reviewed technical conference (Jones & Wang, 2018) and the associated source code is available to everyone at (Jones, 2017b, 2017a).

A Method To Collect And Normalize Functional Classification Information From Publicly Available Resources

As it will be discussed in depth in chapter 3, supervised machine learning classifiers must be trained on known classified samples. Therefore, classifications and samples must be sourced for the classifier's training phase in Figure 3. Publicly available data from resources such as VirusTotal ("VirusTotal," n.d.-a) generally contain classifications developed by the authors of those anti-virus tools and is usually not consistent between vendors. For example, Microsoft may classify a malware sample by the functional characteristics, such as "Ransomware", rather than another vendor that may classify the same sample by the family name, such as "Locky". This dissertation will present a method to collect and normalize publicly available malware classifications so that the information can be used to train Malgazer. The classification information used by Malgazer will be the functional classification, such as "Ransom", but other classification methods could be studied in the future, such as family name. The method of collecting and normalizing the functional classification information is a design science method artifact and the associated source code is available to everyone at (Jones, 2017b). The entropy data sets are available through instructions at the same repository. Users looking for more information about the repositories may find the tour in the appendix useful.

A Method To Scale The Training Of Machine Learning Based Classifiers Using Cloud Resources

Since this dissertation was developed by a single individual with limited time, computing, and financial resources; a method was developed to scale training computations for the machine learning classifiers given those constraints. Scaling these computationally intensive processes across cloud resources was one way to reduce the cost and time involved with computationally

intensive batch jobs, such as training classifiers. While not novel in a general sense, this dissertation will present a programmatic method to create Malgazer training resources on demand, and those resources were used to create the Malgazer classifiers later in this dissertation. The method of scaling computations is presented because it is specific to the malware classification problem, and the method was not clearly mentioned in prior literature. The scaling method was developed such that a single individual can perform the same calculations discussed in this dissertation with minimal costs. Scaling the development of malware classifiers in the cloud is a design science instantiation artifact, and was the sole reason one individual could complete the research described in this dissertation within a year. The process of scaling computations is discussed further in chapter 6.

An Optimized Machine Learning Based Malware Classifier

The values of set "C" will be determined using industry accepted machine learning algorithms to programmatically search for the best parameters for set "C" given the input feature set and error requirements. In machine learning, this process is named "grid searching" for the parameters ("Scikit Learn GridSearchCV," n.d.). Once the candidates for "C" based on running window entropy are selected, Malgazer will be instantiated, in design science terminology. Malgazer will be instantiated using standard Python classes designed to be portable across different types of platforms. The trained classifier is a design science instantiation artifact and is available to everyone at (Jones, 2017b).

A User Application To Automate The Machine Learning Based Malware Classifier

A classifier alone will still require human intervention within the Python interpreter, so another design science artifact will be created as an application so that the user can easily submit malware samples to Malgazer. The application will be developed with Python and function as a web application, complete with programmatic API access. This application will be developed so little human intervention or understanding is required to use the classifier. With this application, the user can interact with Malgazer through a standard web browser or standard command line tools without requiring deep programming knowledge. With one command line, the user is able to instantiate the Malgazer web application on any platform using the open source Docker

application (Docker Inc., n.d.). The user application is a design science instantiation artifact and is available to everyone at (Jones, 2017b).

Assumptions, Scope, And Limitations

There are assumptions made with regard to Malgazer's development. First, it is assumed that the samples presented to Malgazer are already proven to be malware. It would not make sense to pass a benign sample to a malware classifier, as the output would be an unpredictable malware classification while the sample was truly benign. Second, it is assumed that only Microsoft Windows PE files are examined. Each file format could be represented differently as running window entropy, therefore this study could be applied to each of the possible malware executable file types, including weaponized documents, in the same manner as it is applied to Microsoft Windows PE files here.

There are also limitations to Malgazer. First, the biggest limitation is the depth and veracity of the training data set. A classifier built upon bad or biased training data may have unpredictable effects on the classification of unknown samples. Since only one individual worked on this data with limited time, financial, and computing resources instead of a larger data science organization, a manageable number of 60,000 malware samples were collected. In addition, the number of functional classifications were limited to six to make this research project feasible for a single individual. The six functional classifications of Virus, Worm, Backdoor, Trojan, Ransom, and Potentially Unwanted Applications (PUA) were determined by selecting the top functional categories represented by a VirusTotal Intelligence stream in the month of June 2018, as discussed in the qualitative data exploration chapter 5. To prevent a classifier bias, the six groups were represented equally in the training data set using a series of filters at VirusTotal. These filters will be discussed further in chapter 3, research methodology. From the 60,000 samples collected, there were 10,000 samples in each of the six categories. This may not accurately reflect the composition of each classification in the real world overall, but it eliminated the complexity inherent to machine learning based classifiers when using unbalanced training data sets.

Background

This rest of this chapter will explain the background necessary to fully appreciate the concepts this dissertation will examine. A reader already versed in malware analysis and machine learning may wish to skip to the next chapter. This chapter will continue with a section about the use of entropy during malware analysis. The basis of the Malgazer classifier is running window entropy (RWE) data, so it is necessary to understand why entropy is important in the context of malware analysis. Next, this chapter will examine how to perform classification with machine learning, at a generic level. Understanding components of machine learning classification at a generic level is a requirement before applying classification algorithms to malware samples. The concepts introduced in this chapter will be used to help explain additional concepts presented in the literature review chapter, next.

Entropy and Malware Analysis

Basically, entropy is a measure of information encoded in a series of values based upon the probability of those values appearing. Another way to look at entropy is that it is a measure of randomness. Values with higher variance will increase the entropy. Values with no variance will have the minimum entropy of zero. Therefore, entropy can be used to locate more interesting data based upon the magnitude of the entropy value.

"Interesting data" is a subjective description of the data one is looking for. Here, "interesting" is with respect to malware analysis. There are two cases where higher entropy data may be interesting to a malware analyst. First, it is well known that encrypted data is usually higher in entropy than plain text data, and cryptographic keys are usually random-like so they will have high entropy (Stallings, 2017). Calculating the entropy of a file encrypted with strong encryption will lead to a higher entropy value than the same calculation for a plain text version of the same file. If an analyst wanted to locate all of the encrypted files out of a set that were not encrypted, one method they could use is to calculate the entropy of each file and rank them in descending order.

However, this method would not guarantee that only the encrypted files are at the top of the order. Entropy also increases when compression is applied to data. Since compression reduces the data's size, it must reencode the original file differently such that each compressed

value contains more information than the original values it replaced. This will allow the same data to occupy less space, which leads to compression. This information gain per byte from compression leads to more random-like data, therefore the entropy value for compressed data also increases.

Luckily, encrypted and compressed data are two types of data that are of interest to the malware analyst. For example, a malware analyst may want to examine a compressed executable file. These types of executable files typically decompress their payload into memory and run additional commands from there. In this example, the analyst would be looking for areas of the malware sample that were compressed, and those areas could have been identified by the higher entropy.

In another example, a malware analyst may be interested in a section of malware that has been encrypted. Malware authors often employ encryption to slow down analysis and identification of their malware. Encrypted data, like compressed data, will have a higher entropy value. Locating sections within malware samples with higher entropy values is highly likely to lead to sections that are compressed, encrypted, or even a cryptographic key. Even if the sections are not compressed or encrypted, the sections of a sample with higher entropy will still contain more relevant data per byte, in information theory terms. High entropy areas of a malware sample could be the first place an analyst may wish to analyze.

Notice that the examples began with entropy of full files and finished with sections of a malware sample. While malware analysts can measure the entropy of the full malware sample, this information is not always useful. For example, a malware author may measure the entropy of their malware sample and it calculates to “0.9” out of a zero to one scale. The malware author may decide that this entropy value is too high for the malware sample because most security products know of entropy and may use it to detect this new malware sample. The malware author can easily reduce the malware sample’s entropy by appending bytes containing the value zero to the sample. The repetition of the zero value will begin to decrease the full file entropy of the malware sample towards zero. Once enough zero bytes have been appended to lower the entropy to the threshold the author desires, they can stop adding additional bytes. Now, in theory, this malware sample will be less detectable if the detection product factors full file entropy heavily.

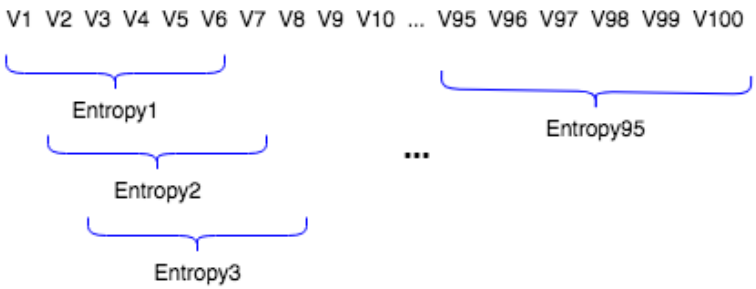


Figure 4. Running window entropy

To combat this entropy manipulation, the security product could examine the last bytes of the file and if they are of low entropy, ignore them. However, the malware author could improve their technique by appending these low entropy bytes to the end of the sections within the executable instead of the file's end. For this malware defense, security products cannot not rely on full file entropy alone to make a malware detection decision. At this point, security products need to examine the entropy from a window of data within the file to make entropy-based decisions. This is where running window entropy can be useful to security products, researchers, and malware analysis.

Running window entropy is entropy of a window within the data stream, and the window is moved to the next byte and calculated again for each value in the stream. The graphical depiction of running window entropy with a window size of six and a malware sample size of one hundred values can be found in Figure 4. Notice that for each byte the window is recalculated for the new values until the window touches the end of the file or data stream. The last entropy data point is "Entropy95", when the right side of the window touches the end of the stream. If the size of the stream is k, and the size of the entropy window is j, the running window entropy vector will contain $k - j + 1$ values. When k and j are equal, there will be only one entropy data point because the entropy window is the same size as the data stream. In this case, it is the same as the full file entropy example previously presented.

For large values of k (the malware size) and j (the entropy window size), there will many calculations taking place. Since the running entropy window moves to the right through the malware data, there will be replicated computations occurring in a double "for" loop. It is possible to reduce the number of replicated computations from this double "for" loop, and

chapter 4 will present one such way to do so. This computation is slow because of the inner log function within Shannon's entropy equation (Shannon, 1948):

$$H = -K \sum_{i=1}^{n} p_i log p_i \quad (1)$$

Here, n is the number of symbols, or 256 since each value within a malware file is a byte, and a byte can have 256 values from zero to 255. Recalculating the log within nested "for" loops becomes a very slow process. Reducing the number of times the log function is called will reduce the overall time it takes to compute the running window entropy data. An optimized, drop-in replacement for the original running window entropy algorithm will be presented in chapter 4. For now, it is important to know that running window entropy is a localized view of an entropy window of size j in the malware file of size k. In the last example of the malware author adding constant bytes to the end of executable sections to lower the entropy of the file, analysis of the running window entropy data could have been used to detect this method.

In Figure 5 and Figure 6, the running window entropy has been plotted for two malware samples. The malware sample A has a SHA256 hash of 00050F30FF95C69F92F 8C32338C4F3473B4F653FF2A1B65375AAC32596A225B4. The malware sample B has a SHA256 hash of 00018690F0A5E684190A95B911E6502658D434249B43EAD719E 49405EB9B9EE7. Without knowing anything else about these two malware samples, it is possible to tell that they must have differences because their entropy graphs are greatly different. The plateaus and valleys in each of the graphs demonstrate structure to the running window entropy data. The structure in one malware sample is not the same structure in the other. It is this running window entropy data that will be used to answer the research question posed previously.

This running window entropy structure could be exploited for machine learning classifications if similarities in the running window entropy data between samples of the same class can be located. To use the running window entropy structure for machine learning classification, the general concepts of machine learning classification must be understood. The

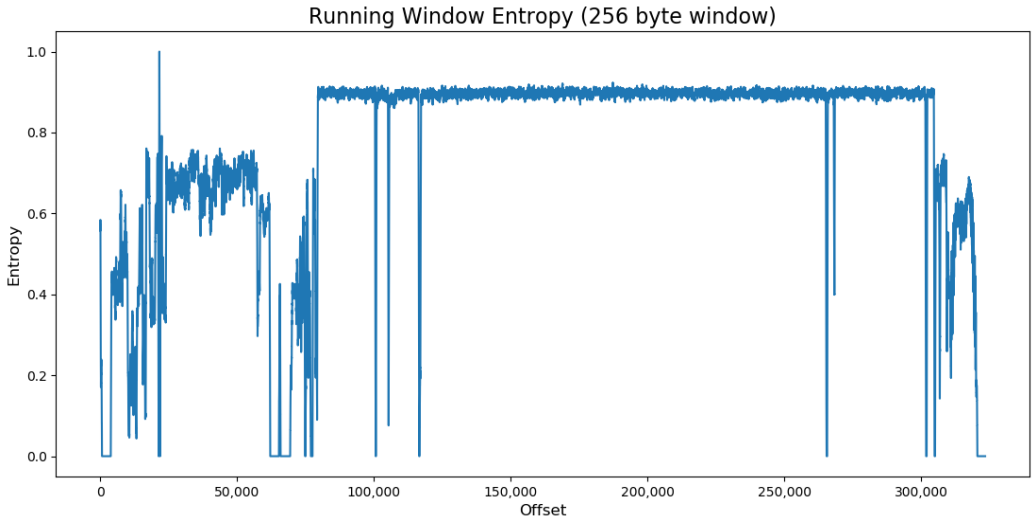


Figure 5. Running window entropy for sample A

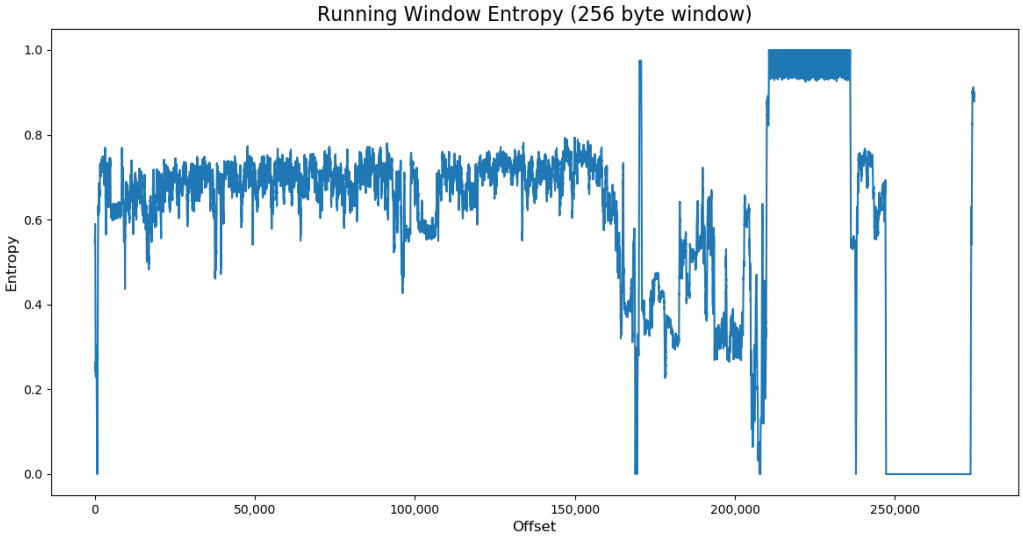


Figure 6. Running window entropy for sample B

next section will discuss how machine learning classification works so that it can be applied to the malware classification problem in this research.

Machine Learning Based Classification

There are numerous uses for machine learning algorithms such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing. This

research will only focus on aspects related to malware classification, so irrelevant concepts such as machine learning regression algorithms will not be covered here. More information on topics not covered here can be found at SciKit Learn's informative website ("scikit-learn," n.d.).

Recall that a classifier is built and operated using the diagram in Figure 3. The input data will be segregated into two groups: a training data set and a testing data set. The training and testing data sets will have the classification information, so Figure 3 represents a supervised machine learning based approach.

There will be a number of machine learning algorithms used throughout this research. Some of the algorithms presented will be used for data preprocessing, while others will be used to select the best machine learning models for a given data set. For preprocessing, algorithms can be used to scale and encode data so that the resulting computations are more manageable. For model selection, there are algorithms that can compare the results of one machine learning model against another and provide the parameters for the model with the best accuracy.

Classification algorithms contain parameters that are "learned" from the input training data set. These parameters constitute the classifier and are fixed after the learning phase so that they can be used to predict the classifications of future malware samples. Hyper-parameters of machine learning classification models are parameters that are not learned from the data but can be tuned by the user. Hyper-parameters can be programmatically determined by the user through the grid searching algorithm, which is an example model selection algorithm mentioned previously.

The core of the classification algorithms within Malgazer will be the following six types of supervised classifiers:

- Decision Trees (DT)
- Random Forests (RF)
- Support Vector Machines (SVM)
- Naïve Bayes (NB)
- K-Nearest Neighbors (KNN)
- Nearest Centroid (NC)

Other algorithms can improve the classification algorithms listed above by "boosting" them. For example, AdaBoost and OneVRest (OvR) both use a series of underlying classifiers chosen from the list above and improves the overall accuracy of the original classifier with a series of smaller classifiers. The boosting algorithms will be considered during this research project.

Artificial intelligence has provided a collection of classification algorithms called "neural networks" that mimic a brain by providing pseudo-neurons that learn classifications based upon the training data. There are two main types of neural networks examined in this research: artificial neural networks (ANN) and convolutional neural networks (CNN). Each of the machine learning classification algorithms will be presented in a subsection below after feature scaling, encoding data, and grid searching are discussed first.

Features Scaling

Features are numerical input values to the classifier. In this research, features will be values from the running window entropy calculations. Since features can contain very large numbers and the machine learning algorithms contain intensive mathematical calculations, it is common to scale the input feature data to make the mathematics more manageable. Scikit learn includes a features scaling algorithm that fits to a training data set ("Scikit Learn StandardScalar," n.d.). Malgazer will use this algorithm to scale the features before inputting them into a model. It is also important to note that there is also an inverse transform available for scaling.

Encoding Data

Classifications, as humans know them, are a description in the form of a string. To take this string and make it something a computer can comprehend, the data can be encoded as a number. For example, the first classification could be "Bot" and the second classification could be "Ransomware". The first classification could be encoded as a zero and the second classification could be encoded as a one. If there were additional classifications, they would be encoded as the next available integer (two, three, four, …). A classification label encoder that transforms the label into a number is implemented in Scikit Learn ("SciKit Learn LabelEncoder," n.d.).

Although a computer can understand an integer more readily than it can understand a string, a computer does not know if that value represents a floating-point continuous value or it is an integer representing a classification. In the example above, it would not make sense to classify a malware sample as "0.7", because it is neither a "Bot" (a zero) or a "Ransomware" sample (a one). To properly represent the classification label data, it can be further encoded as categorical, or sometimes referred to as "one hot", to contain only discrete binary data. After encoding, each value will be transformed from an integer into a binary vector, where the "1" in the binary vector full of zeroes is in the index represented by integer's number. For example, a category of zero would be "0001". A category of one would be "0010". A category of two would be "0100", and so on. It is called "one hot" encoding because the one is enabled in the index represented by the integer value, so one category is represented by one index (or column) in the encoding. A one hot encoder is provided by Scikit Learn ("Scikit Learn OneHotEncoder," n.d.). A similar function is provided as "to_categorical" in the Keras machine learning library ("Keras," n.d.). Both methods achieve the same results. The inverse of this encoding simply locates the index where there is a "1" and reports that index number.

Cross Fold Validation (CFV)/Cross Validation (CV)

Cross fold validation, sometimes called k-fold validation or cross validation, is the process of segmenting the training data set into k equal sized groups and training the classifier on the first $k-1$ groups of the training data, then test the classifier on group k. After that accuracy is computed, the k groups of data will be shifted such that each subset ends up in the testing group once. Once the k calculations have been performed, each accuracy will be averaged to produce the cross validated accuracy for k groups. This process performs k tests on the input data to more accurately reflect the potential real-world performance rather than just one classifier implementation that may have luckily chosen the right samples for training. Cross validation can be performed using functionality from Scikit Learn, or it can be manually implemented and customized with additional features as in Malgazer's case.

Grid Searching

Each of the classification algorithms presented in this chapter have hyper-parameters. Hyper-parameters are parameters that are not automatically learned by the machine learning

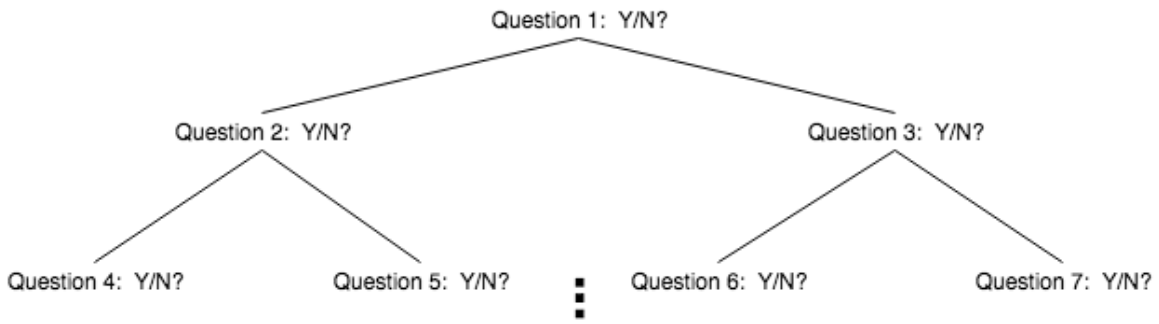


Figure 7. A decision tree classifier

classification model, but they can still be modified by the user. Well-tuned hyper-parameters can increase a model's accuracy, and conversely a hyper-parameter that is not tuned correctly can lead to lower accuracy. It is possible to search through a list of candidate values for each hyper-parameter and select the most performant model through a process called "grid searching". Grid searching has been implemented in Scikit Learn ("Scikit Learn GridSearchCV," n.d.).

Decision Trees (DT)

Decision trees (DT) are a type of classifier. Decision trees involve a series of "if" statements, connected from the top down like a tree, as shown in Figure 7. The "if" statements in the tree are "learned" from the training data to produce the most accurate classifications. Decision trees are implemented in Python through the Scikit Learn package ("Scikit Learn DecisionTreeClassifier," n.d.). There is one main hyper-parameter for this algorithm called "criterion". This parameter is used to measure the quality of the splits in the tree. "Criterion" will be used as a hyper-parameter for the grid searching model selection algorithm later in this research.

Random Forests (RF)

A random forest (RF) classifier is an ensemble of decision tree classifiers that are created from portions of the training data. The individual decision tree classifiers are averaged to provide higher accuracy for the forest. Random forests are implemented in Python through the Scikit Learn package ("Scikit Learn RandomForestClassifier," n.d.). There are two main hyper-parameters for this algorithm called "criterion" and "n_estimators". The first is the criteria for the decision tree classifier, discussed previously. "n_estimators" is the number of decision trees

used in the forest. Both hyper-parameters will be used in the grid searching model selection algorithm later in this research.

Support Vector Machine (SVM)

Support vector machines (SVM) can provide classification by fitting a kernel around the classes in multi-dimensional space. This algorithm is one of the more complex machine learning algorithms and contains many hyperparameters such as kernel, degree, gamma, etc. Scikit Learn has an implementation of the SVM classifier ("Scikit Learn SVC," n.d.) and presents all available hyper-parameters. These hyper-parameters will be used in the grid searching model selection algorithm later in this research.

Naïve Bayes (NB)

The Naïve Bayes (NB) classifier uses the Bayes theorem and the dependence between features to determine classifications. Scikit Learn has an implementation of the Naïve Bayes classifier ("Scikit Learn GaussianNB," n.d.).

K-Nearest neighbors (KNN)

The k-nearest neighbors (KNN) classifier places each training data point in a multi-dimensional space and determines the class of the k-nearest neighbors. Scikit Learn has an implementation of k-nearest neighbors ("Scikit Learn KNeighborsClassifier," n.d.). The most useful hyper-parameter for this algorithm is "n_neighbors", the number of neighbors to use for new classification predictions. This hyper-parameter will be used in the grid searching model selection algorithm later in this research.

Nearest Centroid (NC)

The nearest centroid (NC) algorithm is similar to the KNN algorithm in that it plots the data points in a multi-dimensional space, but it also calculates the "center" of each class and classifies new data based upon the smallest distance to the class's center. Scikit Learn has an implementation of nearest centroid ("Scikit Learn NearestCentroid," n.d.). The nearest centroid algorithm has two hyper-parameters: a metric for calculating distances and a shrink threshold

value. These hyper-parameters will be used in the grid searching model selection algorithm later in this research.

AdaBoost

AdaBoost is not a classifier directly, it is an algorithm that constructs multiple classifiers from a base classifier type. The ensemble of classifiers is created with concentration on the more difficult classifications until the best accuracy is reached. The base classifier could be any of the classifiers discussed previously, such as DT, RF, SVM, and so on. AdaBoost is implemented in Scikit Learn ("Scikit Learn AdaBoostClassifier," n.d.). The important hyper-parameters to AdaBoost are the number of estimators, the learning rate, the algorithm used, and the base estimator type. These hyper-parameters will be used with the grid searching model selection algorithm later in this research.

OneVRest (OVR)

The OneVRest (OVR) classifier is also not a classifier directly, it is an algorithm that constructs multiple classifiers from a base classifier to singly classify each class. As an example, if the classes were "Bot", "Ransom" and "Worm", the OVR classifier would construct a classifier to predict the "Bot" class or not, then it will construct a classifier to predict the "Ransom" class or not, then it will construct a classifier to predict the "Worm" class or not. Each classifier becomes a "this class versus all else" classifier. This is also known as "one vs. rest" or "one vs. all". The OVR classifier is implemented in Scikit Learn ("Scikit Learn OneVsRestClassifier," n.d.). The important hyper-parameter to OVR is the base estimator type. This hyper-parameter will be used with the grid searching model selection algorithm later in this research.

Artificial Neural Networks (ANN)

Artificial Neural Networks (ANN) are layered networks of units where all units in a layer are connected to units in the layer to the left and the right of the layer. A unit creates an output depending on the unit's inputs and parameters. The connections between units are sometimes called "weights". An example ANN is presented in Figure 8. The larger arrows represent

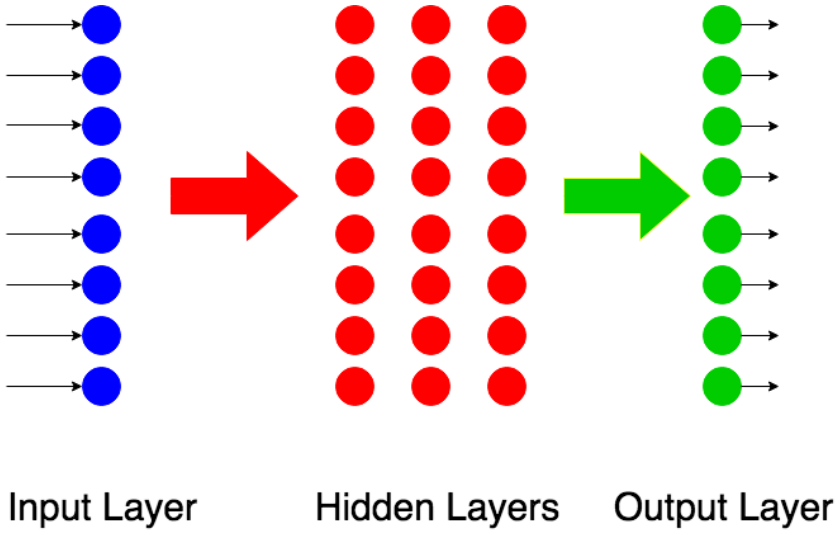


Figure 8. A generic artificial neural network

information flowing forward, and all layers are connected to the layers on the left and right side of that layer. The layers do not all have to be the same size. In Figure 8, each layer has eight units, but each of the hidden layers could have any number of units, making infinitely possible configurations of a neural network. The input layer has the same number of units as the input feature data. The output layer has the same number of units as the number of classifications in the classifier.

When ANNs are trained, as discussed previously in Figure 3, the feature information is fed into the leftmost input layer one observation at a time. With supervised learning, classifications are available in the training data so that the true classification information can be compared to the predicted classification at the rightmost output layer. Since the classification data has been preprocessed using label and one hot encoding, each classification is represented by one output node in the output layer. The learning process for an ANN is to supply a feature observation at the input layer and compare the predicted classification with the known classification on the output layer, noting the error in the prediction to the true classification. This process is repeated for all observations in the training data. Then, through a process known as gradient descent, the parameters for each of the units in the network are modified in an attempt to reduce the overall classification error. Each time the training set is calculated, it is called an epoch. After numerous epochs the classification error should begin to converge. It is at this point that the network has "learned" the classifications of the training data. If the error does not converge, the network did not "learn" the classifications adequately.

ANNs have many hyper-parameters if the structure of the network is considered. Other hyper-parameters exist in the form of metadata such as activation functions or how the descent is optimized. There are many different methods available for ANN implementation, but the one used for this dissertation was the Python Keras library running with a Tensorflow backend ("Keras," n.d.; "Tensorflow," n.d.).

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are a neural network because of the layers and connections, but there are different types of layers available in addition to the ANN units. Instead of simple connections between layers, layers can perform convolutions and pooling. Convolutions can be thought of as filters while pooling is similar to down sampling. CNNs are more complex than ANNs and therefore require more time for processing. However, the tradeoff is that CNNs can classify interdependent multi-dimensional data typically better than ANNs. CNNs are regularly used in handwriting recognition in two-dimensional image data, for example. A one-dimensional array representing the running window entropy of a malware sample is similar to a one-dimensional array of audio signal. Since CNNs are used to classify audio signals as speech, the same type of approach could be used to match the structures within the running window entropy of malware samples to their respective classifications.

CNNs have similar hyper-parameters to ANNs with respect to the structure and types of units used in the network. There are many different methods available for CNN implementation, but the one used for this dissertation is the Python Keras library running with a Tensorflow backend ("Keras," n.d.; "Tensorflow," n.d.).

Summary

In summary, the reader should expect this dissertation to examine automated machine learning based malware classifiers built from the running window entropy feature space. This classifier, named herein as Malgazer, will not only be compared to other malware classifiers from prior research theoretically, but it will also be compared empirically to a method from prior research using GIST features. In order to complete this comparison, the prior literature will be categorized and compared at the component level in the literature review chapter using the

generalized malware classification model presented in Figure 2 and Definition 1. Malgazer will be developed using the model in Figure 2 and Definition 1 using a methodical searching algorithm to find the appropriate values for the set "C". The searching and training of Malgazer classifiers will be performed programmatically and at scale using cloud resources that any individual can execute. Once the Malgazer classifier has been instantiated, the classifier will be available to users through a web application that can run on nearly any type of platform locally or in the cloud for users without deep programming knowledge.

This chapter also presented the background of the use of entropy for malware analysis and machine learning classification. First, this chapter presented that entropy is a measure of information and running window entropy can be used to locate entropy variations within a data stream. Second, this chapter presented a short introduction to some of the algorithms to preprocess input features and train supervised machine learning classifiers on the training data set. Both of these topics will be used to explain and categorize the prior research in the literature review chapter, next.

CHAPTER 2

LITERATURE REVIEW

Introduction

In this chapter, the prior research on machine learning based malware classifiers is explored. The prior research was first clustered into categories. Next, a chronological overview identifies the components of each prior method using the generalized machine learning based malware classifier model presented in Figure 2 and Definition 1. It was during the literature review phase of this dissertation that the need for this model became apparent so that the prior literature could be compared and explained in an organized fashion. Presenting the prior literature using this model allowed for the comparison of components within prior research, plus it allowed for the comparison of components from prior research to the components of Malgazer. Demonstrating that a substantial amount of prior literature can be described using the generalized machine learning based malware classifier model validates the utility of this design science model artifact.

This chapter begins with an introduction to malware classification method categories and a chronological overview of each prior method surveyed. This chapter concludes with prior literature that was relevant to this dissertation but did not properly fit into the categories because it contained foundational concepts rather than malware classification methods. Based upon the prior literature, Malgazer's contribution to science will be identified in the last section of this chapter.

As noted in the introduction chapter in Figure 1, malware detection is a subset of malware classification. In other words, a malware detector is also a malware classifier in that it only classifies two classes: malware and benign. Therefore, relevant machine learning based malware detection research is included in this chapter for the sake of completeness and detectors are identified as such in the overview.

Malware Classification Method Categories

Several categories were used to cluster the prior research. After a thorough survey of the literature, the classification method categories identified here consist of the method type (detector or classifier), followed by a dash, then the data the method used to make predictions, such as “dynamic analysis” and “images from bytes”. These are the input features. Each method belongs to only one category. Each of the categories developed from the prior literature are identified and briefly described next.

Malware Detector - Boot Sectors

This category represents a malware detection method that uses boot sectors as the input features to a machine learning model.

Malware Detector – Dynamic Analysis

This category represents a malware detection method that uses dynamic analysis data as input features to a machine learning model. Dynamic analysis includes features such as API calls, running time, memory usage, CPU usage, etc.

Malware Detector – Entropy Statistics

This category represents a malware detection method that uses entropy statistical information as input features to a machine learning model.

Malware Detector – Images From Bytes And n-grams

This category represents a malware detection method that uses images created from the sample’s bytes and n-grams of the bytes as input features to a machine learning model. In this category, images are calculated from the raw byte values of the malware sample. n-grams are n-tuple representations of the bytes.

Malware Detector – Opcode Statistics

This category represents a malware detection method that uses statistical information about opcodes from the sample as input features to a machine learning model. Opcodes are the machine codes the processor executes.

Malware Detector – PDF Streams

This category represents a malware detection method that uses PDF streams as input features to a machine learning model. PDF streams are only found in PDF files.

Malware Detector – Static Analysis

This category represents a malware detection method that uses static analysis as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc.

Malware Detector – Static Analysis And Opcodes

This category represents a malware detection method that uses static analysis and opcodes as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc. Opcodes are the machine codes the processor executes.

Malware Detector – Static Analysis And n-grams

This category represents a malware detection method that uses static analysis and n-gram representation of the byte code as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc. n-grams are n-tuple representations of the bytes.

Malware Detector – Static And Dynamic Analysis

This category represents a malware detection method that uses static and dynamic analysis data as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc. Dynamic analysis includes features such as API calls, running time, memory usage, CPU usage, etc.

Malware Detector – Structural Entropy

This category represents a malware detection method that uses structural entropy as input features to a machine learning model. Note that structural entropy is similar to running window entropy.

Malware Detector – n-grams

This category represents a malware detection method that uses byte code n-grams as input features to a machine learning model. n-grams are n-tuple representations of the bytes.

Malware Classifier – Byte Code

This category represents a malware classification method that uses byte code as input features to a machine learning model. The byte code is the malware sample's raw byte values.

Malware Classifier – Dynamic Analysis

This category represents a malware classification method that uses dynamic analysis data as input features to a machine learning model. Dynamic analysis includes features such as API calls, running time, memory usage, CPU usage, etc.

Malware Classifier – HMM

This category represents a malware classification method that uses hidden Markov models as input features to a machine learning model.

Malware Classifier – Images From Bytes

This category represents a malware classification method that uses images created from the sample's byte code as input features to a machine learning model. In this category, images are calculated from the raw byte values of the malware sample.

Malware Classifier – Images From Bytes And Opcodes

This category represents a malware classification method that uses images created from the sample's byte code and opcodes as input features to a machine learning model. In this category, images are calculated from the raw byte values of the malware sample. Opcodes are the instructions the CPU executes.

Malware Classifier – Intermediate Language Of Opcodes

This category represents a malware classification method that uses an intermediate language representation of opcodes as input features to a machine learning model. An intermediate language is a language that represents the operation of a program but is at a higher level.

Malware Classifier – Static Analysis

This category represents a malware classification method that uses static analysis data as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc.

Malware Classifier – Static Analysis And Images From Bytes

This category represents a malware classification method that uses static analysis data and images from the sample's byte code as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc. In this category, images are calculated from the raw byte values of the malware sample.

Malware Classifier – Static Analysis And Opcodes

This category represents a malware classification method that uses static analysis data and opcodes as input features to a machine learning model. Static analysis includes features such as file length, number of sections, names of sections, etc. Opcodes are the instructions the CPU executes.

Malware Classifier – Static And Dynamic Analysis

This category represents a malware classification method that uses static and dynamic analysis data as input features. Static analysis includes features such as file length, number of sections, names of sections, etc. Dynamic analysis includes features such as API calls, running time, memory usage, CPU usage, etc.

Malware Classifier – Structural Entropy

This category represents a malware classification method that uses structural entropy as input features to a machine learning model. Note that structural entropy is similar to running window entropy.

Overview Of The Prior Works for Malware Classification

This section provides a chronological overview of seventy-one methods surveyed from prior literature. This overview provides a brief description and will categorize each method using the categories that were described in the previous section. Each method is identified with a unique number and by citation so that the process of referencing the methods by either field is clear.

The overview information for each publication will identify the components in Figure 2 and Definition 1 for each method. Full quotes of the original source were included where appropriate, and paraphrasing was used for the others. Although the component functions (T(), P(), etc.) were not explicitly constructed in this section, the information used to construct such functions is provided. This was purposefully chosen because some of the functions were complex. The explanations were more concise when explained in shorter human concepts or quotations from the original publication rather than any other method explored by this author.

1996

Method #01. (Tesauro, Kephart, & Sorkin, 1996): In 1996, this was one of the first papers that used machine learning algorithms to classify malware. This research provided a detector for boot sector viruses. The detector used trigrams of boot sectors for their feature space as input to an artificial neural network (ANN). They reported an accuracy of 80-85% for viral boot sectors.

- CATEGORY: Malware Detector – Boot Sectors
- EFFICACY: accuracy 80-85% for viral boot sectors
- TRAINING DATA: 200 boot sector viruses and 100 legitimate boot sectors.
- TRANSFORMATION FUNCTION T(): Calculates trigrams of boot sector viruses. Eliminates any trigrams from valid boot sectors. Eliminates trigrams that are "too frequent". Eliminates trigrams that are not part of a 4-cover of available trigrams in their training set. This, according to the authors, leaves only approximately 50 trigrams that are features.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Artificial Neural Network with no hidden layers. Has a tunable parameter for threshold. This paper discusses a threshold of 0.5-0.7.
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Can tune the threshold up to 0.7 to have "an adequate safety cushion for false positives, while only slightly increasing (by a few percent) the false positive rate".
- POST PROCESSING FUNCTION PP(): Not available

2001

Method #02. (Schultz, Eskin, Zadok, & Stolfo, 2001): In 2001, this paper introduced used a malware detector that used three machine learning algorithms on features measured from malware static analysis. They reported an accuracy of 97.76%.

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy 97.76%
- TRAINING DATA: “4,266 programs split into 3,265 malicious binaries and 1,001 clean programs”.
- TRANSFORMATION FUNCTION T(): "We used system resource information, strings and byte sequences that were extracted from the malicious executables in the data set as different types of features." Three DLL features were extracted: "1. The list of DLLs used by the binary 2. The list of DLL function calls made by the binary 3. The number of

different function calls within each DLL". Strings were extracted. Byte sequences were extracted.

- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Three learning models were used: "an inductive rule-based learner that generates boolean rules based on feature attributes", "a probabilistic method that generates probabilities that an example was in a class given a set of features" and "a multi-classifier system that combines the outputs from several classifiers to generate a prediction." "RIPPER, Naive Bayes, and a Multi-Classifier system" were used.
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

2004

Method #03. (Kolter & Maloof, 2004): In 2004, this paper used several machine learning algorithms on n-gram features of the malware sample's bytes to detect malware. They reported an area under the ROC curve (AUC) of 0.996. This paper was republished in 2006 (Kolter & Maloof, 2006).

- CATEGORY: Malware Detector – n-grams
- EFFICACY: AUC of 0.996
- TRAINING DATA: "We gathered 1971 benign and 1651 malicious executables".
- TRANSFORMATION FUNCTION T(): "We used the hexdump utility [29] to convert each executable to hexadecimal codes in an ascii format. We then produced n-grams, by combining each four-byte sequence into a single term. For instance, for the byte sequence ff 00 ab 3e 12 b3, the corresponding n-grams would be ff00ab3e, 00ab3e12, and ab3e12b3. This processing resulted in 255,904,403 distinct n-grams… We then selected the top 500 n-grams, a quantity we determined through pilot studies (see Section 6.1)"
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "ibk, tfidf, naive Bayes, support vector machines (svms), decision trees, boosted naive Bayes, boosted svms, and

boosted decision trees… several learning methods, most of which are implemented in weka: ibk, tfidf, naive Bayes, a support vector machine (svm), and a decision tree."

- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): The collection of the "relevant" n-grams
- POST PROCESSING FUNCTION PP(): Not available

2008

Method #04. (Siddiqui, 2008): In 2008, Siddiqui published a thesis that introduced a malware detector using a variety of machine learning algorithms on features produced from the n-grams of byte code and static analysis of the sample. Siddiqui reported an accuracy as high as 98.4%.

- CATEGORY: Malware Detector – Static Analysis and n-grams
- EFFICACY: accuracy as high as 98.4%
- TRAINING DATA: Approximately 8,000 samples with 4,617 malicious binaries.
- TRANSFORMATION FUNCTION T(): T() can calculate n-grams, api/system calls, assembly instructions, and hybrid features.
- PRE-PROCESSING FUNCTION P(): A preprocessing function that limited the numbers of features based upon Chi-Squared tests was used in this research.
- MACHINE LEARNING MODEL FUNCTION MLM(): Logistic regression, neural network, decision tree, support vector machines, bagging, random forest, and principal component analysis
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): A tuning function selecting the most "useful" MLM was used.
- POST PROCESSING FUNCTION PP(): Not available

Method #05. (Perdisci, Lanzi, & Lee, 2008a): In 2008, this paper introduced a malware detector for packed samples. The features from static analysis of the sample were provided to numerous machine learning models to detect packed samples. They reported an accuracy of 98.91% and an AUC of 0.9995.

- CATEGORY: Malware Detector – Static Analysis

- EFFICACY: accuracy of 98.91% and an AUC of 0.9995
- TRAINING DATA: "We performed experiments on 5,498 executables. 2,598 are packed computer viruses collected from the Malfease Project dataset (http://malfease.oarci. net), 2,231 are benign executables extracted from a clean installation of Windows XP Home plus several common user applications, and 669 are packed benign executables obtained by manually packing applications selected from the start menu of Windows XP using 17 different executable packing tools."
- TRANSFORMATION FUNCTION T(): The nine features are measured as: the number of standard and non-standard sections; the number of executable sections; the number of readable/writeable/executable sections; the number of entries in the IAT; the PE header, code, data, and file entropy.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Naïve Bayes, J48 decision tree, k-nearest-neighbors, Multi Layer Perceptron (MLP) ("constructed using 1 input layer with 9 nodes (one for each feature), 1 hidden layer with 5 nodes, and 1 output layer with 2 nodes (one for each class). All the hidden and output nodes use a sigmoidal activation function, whereas the input nodes use a linear activation function. The backpropagation algorithm was used for training, with 20% of training patterns reserved for validation."), entropy threshold classifier.
- ERROR FUNCTION E(): ROC curves
- TUNING FUNCTION TU(): For the entropy threshold classifier, the threshold can be tuned.
- POST PROCESSING FUNCTION PP(): Not available

Method #06. (Perdisci, Lanzi, & Lee, 2008b): In 2008, an additional paper was published from the authors of (Perdisci et al., 2008a). The second paper included n-gram analysis in the feature space. They reported an accuracy of 87.3% and AUC of 0.977.

- CATEGORY: Malware Detector – Static Analysis and n-grams
- EFFICACY: accuracy of 87.3% and an AUC of 0.977
- TRAINING DATA: "We developed a proof-of-concept version of McBoost and evaluated it on 5,586 malware and 2,258 benign programs."

- TRANSFORMATION FUNCTION T(): There are two classifiers proposed in this research. The first's features are: "We extract nine features from the PE file, namely: 1) Number of standard sections; 2) Number of non-standard sections; 3) Number of Executable sections; 4) Number of Readable/Writable/Executable sections; 5) Number of entries in the IAT; 6) Entropy of the PE header; 7) Entropy of the code (i.e., executable) sections; 8) Entropy of the data sections; 9) Entropy of the entire PE file." The second set of features, for the detector, are unpacked version of the input binaries.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Naïve Bayes, J48 decision tree, k-nearest-neighbors, Multi Layer Perceptron (MLP) ("constructed using 1 input layer with 9 nodes (one for each feature), 1 hidden layer with 5 nodes, and 1 output layer with 2 nodes (one for each class). All the hidden and output nodes use a sigmoidal activation function, whereas the input nodes use a linear activation function. The backpropagation algorithm was used for training, with 20% of training patterns reserved for validation."), entropy threshold classifier. Additionally, a Bagged-Decision-Threes (BDT) was used as the malware detector.
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): For the entropy threshold classifier, the threshold can be tuned. There is also a decision threshold that can be set.
- POST PROCESSING FUNCTION PP(): The post processing function of the output from the packer detector is used to either unpack the sample for n-gram analysis, or use the binary as is.

2009

Method #07. (Jason L Wright, 2009): In 2009, Wright introduced a method for detecting functional blocks as cryptography using opcode statistics as features to a neural network. Wright reported an accuracy above 99%.

- CATEGORY: Malware Detector – Opcode Statistics
- EFFICACY: accuracy above 99%
- TRAINING DATA: The C library from OpenBSD.

- TRANSFORMATION FUNCTION T(): "For NNLC, the inputs to the artificial neural network (ANN) are the total number of instructions with the following opcodes: XOR (exclusive or), SHL (logical shift right), SHL (logical shift left), ROR (rotate right), and ROL (rotate left). Additionally, the densities of these instructions are applied to five more inputs. Density is defined here as the number of the specific opcodes divided by the total number of opcodes in the function."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Various architectures were tried for the neural network. The architecture that yielded the lowest total error consisted of 10 input neurons, 5 neurons in a single hidden layer, and a single output neuron."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Can modify the hyperbolic tangent gain and learning constant.
- POST PROCESSING FUNCTION PP(): Not applicable

Method #08. (M. Shafiq, Tabish, & Farooq, 2009): In 2009, this paper introduced a malware detector based upon static analysis features using decision trees and neural networks. They reported a detection rate of 0.996.

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy of 99.6%
- TRAINING DATA: "We have obtained about half a million malicious executables from Offensive Computing.net. We have also obtained several thousand benign executables from freshly installed Windows machines and online resources such as sourceforge and CNET's download.com."
- TRANSFORMATION FUNCTION T(): Number of standard, non-standard, executable, read/write/executable sections; number of entries in IAT; entropy of PE header, code sections, data sections, entire PE file; COFF file header; optional header; text section; data section; resource section; resource enumeration.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): A neural network and J48 decision trees.

- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #09. (M. Z. Shafiq, Tabish, Mirza, & Farooq, 2009b): In 2009, this paper introduced a malware detector based upon static analysis features using a variety of machine learning algorithms. They reported an accuracy of more than 99%. This paper was very similar to (M. Z. Shafiq, Tabish, Mirza, & Farooq, 2009a).

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy of more than 99%
- TRAINING DATA: "The first malware dataset is the VX Heavens Virus collection consisting of more than ten thousand malicious PE files. The second malware dataset is the Malfease dataset, which contains more than five thousand malicious PE files. We also collected more than one thousand benign PE files from our virology lab, which we use in conjunction with both malware datasets in our study."
- TRANSFORMATION FUNCTION T(): DLLs referred; COFF file header; optional header - standard fields; optional header - Windows specific fields; optional header - data directories; .text section - header fields; .data section - header fields; resource directory table & resources.
- PRE-PROCESSING FUNCTION P(): Redundant feature removal (RFR), principal component analysis (PCA), Haar wavelet transform (HWT)
- MACHINE LEARNING MODEL FUNCTION MLM(): Instance based learner Ibk, decision tree J48, Naïve Bayes, support vector machines using sequential minimal optimization (SMO), inductive rule learner RIPPER
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #10. (Tabish, Shafiq, & Farooq, 2009): In 2009, this paper introduced a malware detector based upon n-grams of the malware's byte code. They reported an accuracy of more than 90%.

- CATEGORY: Malware Detector – n-grams

- EFFICACY: accuracy more than 90%
- TRAINING DATA: "We have collected 1, 447 benign PE files from the local network of our virology lab. The collection contains executables such as Packet CAPture (PCAP) file parsers compiled by MS Visual Studio 6.0, compressed installation executables and MSWindows XP/Vista applications' executables. The diversity of the benign files is also evident from their sizes, which range from a minimum of 4 KB to a maximum of 104, 588 KB (see Table 3). Moreover, we have used two malware collections in our study. First is the VXHeavens Virus Collection, which is labeled and is publicly available for free download. We only consider PE executables to maintain focus. Our filtered dataset contains 10, 339 malicious PE files. The second dataset is the Malfease malware."
- TRANSFORMATION FUNCTION T(): DLLs referred; COFF file header; optional header - standard fields; optional header - Windows specific fields; optional header - data directories; .text section - header fields; .data section - header fields; resource directory table & resources
- PRE-PROCESSING FUNCTION P(): Redundant feature removal (RFR), principal component analysis (PCA), Haar wavelet transform (HWT)
- MACHINE LEARNING MODEL FUNCTION MLM(): Instance based learner Ibk, decision tree J48, Naïve Bayes, support vector machines using sequential minimal optimization (SMO), inductive rule learner RIPPER
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #11. (Tabish et al., 2009): In 2009, this paper introduced a malware detector that used n-grams of a malware sample's bytes for several machine learning model algorithms. They reported an accuracy of more than 90%.

- CATEGORY: Malware Detector – n-grams
- EFFICACY: accuracy more than 90%
- TRAINING DATA: "We have used 'VX Heavens Virus Collection' database which is available for free download in the public domain... This is a comprehensive database that contains a total of 37,420 malware samples. The samples consist of backdoors,

constructors, flooders, bots, nukers, sniffers, droppers, spyware, viruses, worms and trojans etc. We only consider Win32 based malware in PE file format. The filtered dataset used in this study contains 10, 311 Win32 malware samples."

- TRANSFORMATION FUNCTION T(): Samples are reduced to blocks of bytes.
- PRE-PROCESSING FUNCTION P(): "Overall, the features' set consist of 13 diverse features, which are separately computed on 1, 2, 3, and 4 gram frequency histograms. This brings the total size of features' set to 52 features per block." Features include: Simpson's Index, Canberra Distance, Minkowski Distance of Order, Manhattan Distance, Chebyshev Distance, Bray Curtis Distance, Angular Separation, Correlation Coefficient, Entropy, Kullback-Leibler Divergence, Jensen-Shannon Divergence Itakura-Saito Divergence, Total Variation
- MACHINE LEARNING MODEL FUNCTION MLM(): "We have used Instance based (IBk), Decision Trees (J48), Naïve Bayes and the boosted versions of these classifiers in our pilot study. For boosting we have used AdaBoostM1 algorithm. We have utilized implementations of these algorithms available in WEKA."
- ERROR FUNCTION E(): ROC curves
- TUNING FUNCTION TU(): The block size, in bytes
- POST PROCESSING FUNCTION PP(): A correlation module that bases detections on MLM().

2010

Method #12. (J L Wright & Manic, 2010): In 2010, Wright continued the research into detection of functional blocks of cryptography. Wright reported an accuracy of 99.5%.

- CATEGORY: Malware Detector – Opcode Statistics
- EFFICACY: accuracy of 99.5%
- TRAINING DATA: The C library from OpenBSD.
- TRANSFORMATION FUNCTION T(): "The input data set for training the networks consists of the count of the number of each distinct processor opcode in each function in a library as well as the density of each opcode."
- PRE-PROCESSING FUNCTION P(): Not available

- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Cross-fold validation is used to determine the best structure for the ANN.
- POST PROCESSING FUNCTION PP(): Not available

Method #13. (Sun, Versteeg, Boztas, & Yann, 2010): In 2010, this paper introduced a packer classifier. Packers are a type of executable compression and are often used with malware samples by their authors. This classifier used structural entropy as a feature space and they reported an accuracy of greater than 99%.

- CATEGORY: Malware Classifier – Structural Entropy
- EFFICACY: accuracy of greater than 99%
- TRAINING DATA: "We test these algorithms on a large data set that consists of clean packed files and 17,336 real malware samples."
- TRANSFORMATION FUNCTION T(): "In order to extract the best features from packed files, we employ a refined version of the sliding window randomness test with trunk pruning method [5]. Compare with the original sliding window randomness test (see Appendix B), the window size and skip size used in this research are set to the same value. That is, there is no overlapping windows. Therefore, no repeat information is used in the feature set."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "We test various statistical classification algorithms, including k−Nearest Neighbor, Best-first Decision Tree, Sequential Minimal Optimization and Naïve Bayes."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Window and skip sizes for entropy window.
- POST PROCESSING FUNCTION PP(): Not available

2011

Method #14. (L Nataraj, Karthikeyan, Jacob, & Manjunath, 2011): In 2011, this paper introduced a malware classifier that used bytes from the malware samples represented as two-

dimensional images as features. The images were filtered using an algorithm known as "GIST" to produce 320 features to the KNN machine learning classification algorithm. They reported an accuracy of 98%. This method is the method that will be compared empirically with Malgazer as this author has been in contact with one of the authors from this paper. This algorithm will be identified throughout this dissertation as the "GIST" method. Additional online references for this method are available at (Laks, 2013, 2014), as indicated by one of the authors of this paper.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: accuracy of 98%
- TRAINING DATA: "25k malware from Anubis and VxHeavens Dataset."
- TRANSFORMATION FUNCTION T(): Bytes from the sample are transformed to a gray scale image.
- PRE-PROCESSING FUNCTION P(): "To compute texture features, we use GIST, which uses a wavelet decomposition of an image. This feature has been successful in scene classification and object classification. Each image location is represented by the output of filters tuned to different orientations and scales. We use a steerable pyramid with 8 orientations and 4 scales applied to the image."
- MACHINE LEARNING MODEL FUNCTION MLM(): "We use k-nearest neighbors with Euclidean distance for classification."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #15. (Rieck, Trinius, Willems, & Holz, 2011): In 2011, this paper introduced a malware classifier that used dynamic analysis information and KNN clustering. They reported an F-measure of 0.96.

- CATEGORY: Malware Classifier – Dynamic Analysis
- EFFICACY: F-measure of 0.96
- TRAINING DATA: "The reference data set is extracted from a large database of malware binaries maintained at the CWSandbox web site. The malware binaries have been collected over a period of three years from a variety of sources, such as honeypots, spam traps, anti-malware vendors and security researchers. From the overall database, we

select binaries which have been assigned to a known class of malware by the majority of six independent anti-virus products... The selected malware binaries are then executed and monitored using CWSandbox, resulting in a total of 3,133 behavior reports in MIST format"

- TRANSFORMATION FUNCTION T(): Malware is executed in a sandbox environment. The system calls create a sequential report, which becomes a vector space. The authors developed a "MIST" language representation of the machine level instructions.
- PRE-PROCESSING FUNCTION P(): The MIST vector space is reduced to real numbers using natural language processing.
- MACHINE LEARNING MODEL FUNCTION MLM(): A k-nearest neighbors clustering algorithm augmented with a classification algorithm based upon the clusters.
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

2013

Method #16. (Dahl, Stokes, Deng, & Yu, 2013): In 2013, this paper introduced a malware detector that used n-grams of the malware's byte code and static analysis of the sample in neural networks. They reported an error rate as low as 0.42%.

- CATEGORY: Malware Detector – n-grams
- EFFICACY: error rate as low as 0.42%
- TRAINING DATA: "To construct the labeled training dataset, we first obtain over 2.6 million files, 1,843,359 of which are malicious and 817,485 of which are benign."
- TRANSFORMATION FUNCTION T(): "In our work, we also employ sparse binary features based on file strings, application programming interface (API) tri-grams, and API call plus parameter value. To achieve good classification accuracy, we use over 179 thousand sparse, binary features generated from feature selection… we extract three types of features including null-terminated patterns observed in the process' memory, tri-grams of system API calls, and distinct combinations of a single system API call and one input parameter."

- PRE-PROCESSING FUNCTION P(): "Enumerating all of the distinct combinations of the three attribute sets yields over 50 million possible features. In order to reduce the input space to a reasonable set of features which can be classified by standard supervised learning techniques, we next perform feature selection using mutual information. Feature selection generated over 179 thousand sparse binary features that best discriminate each class (e.g. malware family, malware, benign) from every other class in our dataset." The high dimensional space is projected into a lower dimensional space to reduce the number of features.
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks and logistic regression
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #17. (Santos, Brezo, Ugarte-Pedrero, & Bringas, 2013): In 2013, this paper introduced a malware detector that used opcode statistics as features in a variety of machine learning algorithms. They reported an accuracy of more than 90%, and an AUC of more than 0.90.

- CATEGORY: Malware Detector – Opcode Statistics
- EFFICACY: accuracy more than 90%, AUC more than 0.90
- TRAINING DATA: "We collected malware from the VxHeavens website to assemble a malware dataset of 13,189 malware executables. This dataset contained only Portable Executable (PE)5 executable files, and it was made up of different types of malicious software (e.g., computer viruses, Trojan horses and spyware). For the benign software dataset, we collected 13,000 executables from our computers. This benign dataset included text processors, drawing tools, windows games, Internet browsers and PDF viewers."
- TRANSFORMATION FUNCTION T(): "We use a representation based on opcodes (i.e., operational codes in machine language). Our method is based on the frequency of appearance of opcode-sequences: it trains several data-mining algorithms in order to detect unknown malware… Therefore, our approach only uses opcodes and we discard

the operands within the instructions." Disassemble the executables, generate the opcode profile, compute the opcode relevance

- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Decision trees, support vector machines (SVM), k-nearest neighbors, Bayesian networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Experimentation on the parameters of the MLM() algorithms.
- POST PROCESSING FUNCTION PP(): Not available

Method #18. (Annachhatre, Austin, & Stamp, 2013): In 2013, this paper introduced a malware classifier based on hidden Markov model scoring and k-means clustering. They did not report an efficacy.

- CATEGORY: Malware Classifier – HMM
- EFFICACY: Not reported
- TRAINING DATA: "The dataset was requested from the Malicia project website. It comprises of 11000+ malware binaries collected from 500 drive-by download servers over a period of 11-months. We have used 9442 malware samples from this dataset for malware classification"
- TRANSFORMATION FUNCTION T(): "We scored the malware samples using the hidden Markov models for variety of compilers and malware generators (GCC, MINGW, CLANG, TURBOC, TASM and MWOR)."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Hidden Markov models and k-means clustering
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

2015

Method #19. (Lakshmanan Nataraj, Karthikeyan, & Manjunath, 2015): In 2015, this paper introduced a malware classifier that used n-grams of the malware sample's bytes, fuzzing hashing, and image similarity to a "Sparse Representation based Classification (SRC) and Nearest Neighbor (NN) Classifier". They reported an accuracy of 98.66%.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: accuracy of 98.66%
- TRAINING DATA: "Two datasets: Malimg and Malheur[:] Malimg Dataset: 25 families, 80 samples per family, M = 840,960. Malheur Dataset: 23 families, 20 samples per family, M = 3,364,864."
- TRANSFORMATION FUNCTION T(): n-grams, fuzzy hashing, image similarity
- PRE-PROCESSING FUNCTION P(): Random projections
- MACHINE LEARNING MODEL FUNCTION MLM(): "Sparse Representation based Classification (SRC) and Nearest Neighbor (NN) Classifier"
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #20. (Wolff & Chisholm, 2015): In 2015, this presentation introduced a structural entropy analysis method and used structural entropy as a feature to a lasso logistic regression machine learning model to detect malware. They reported an accuracy of 87.2%.

- CATEGORY: Malware Detector – Structural Entropy
- EFFICACY: accuracy of 87.2%
- TRAINING DATA: ""Data is a corpus of Windows Portable Executable (PE) files downloaded from Virus Total in June, 2014… We formed a balanced data set of 39,968 files; of these, 19,991 (50.01%) were labeled as malware, whereas 19,997 (49.98%) were labeled as clean computer programs."
- TRANSFORMATION FUNCTION T(): Structural entropy
- PRE-PROCESSING FUNCTION P(): Detrended Fluctuation Analysis (DFA), Mean Changepoint Modeling (MCPM), Wavelet Decomposition, Summary Statistics
- MACHINE LEARNING MODEL FUNCTION MLM(): Lasso logistic regression
- ERROR FUNCTION E(): Confusion matrices

- TUNING FUNCTION TU(): Numerous parameters can be tuned.
- POST PROCESSING FUNCTION PP(): Not available

Method #21. (Nath & Mehtre, 2015): In 2015, this paper introduced a malware detector for PDF streams. The feature space was PDF streams and the machine learning model was decision trees.

- CATEGORY: Malware Detector – PDF Streams
- EFFICACY: Not reported
- TRAINING DATA: "Here, we are creating the dataset by taking a group of benign files and malicious files. We have got almost collected 10982 malicious files along with 173 latest samples belonging to targeted attacks from [20]."
- TRANSFORMATION FUNCTION T(): Extraction of streams from PDFs.
- PRE-PROCESSING FUNCTION P(): Calculation of entropy on extracted data
- MACHINE LEARNING MODEL FUNCTION MLM(): Decision trees
- ERROR FUNCTION E(): Not available
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #22. (Gu et al., 2015): In 2015, this paper introduced a malware detector using structural entropy and wavelet analysis in a "Weighted decision" machine learning algorithm. They reported an accuracy of 93.1%.

- CATEGORY: Malware Detector – Structural Entropy
- EFFICACY: accuracy of 93.1%
- TRAINING DATA: "We use this system to test 200 samples."
- TRANSFORMATION FUNCTION T(): Wavelet decomposition of entropy
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Weighted decision"
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #23. (Saxe & Berlin, 2015): In 2015, this paper introduced a malware detector using entropy statistics and static analysis of the sample with a neural network classifier. They reported an accuracy of 95%.

- CATEGORY: Malware Detector – Entropy Statistics
- EFFICACY: accuracy of 95%
- TRAINING DATA: "Our benign and malware binaries were drawn from Invincea's own computer systems and Invincea's customers networks. We used malicious binaries obtained from both the Jotti commercial malware feed and from Invincea's private malware database. Our final dataset, after VirusTotal filtering, contains 431,926 binaries, with 81,910 labeled as benignware and 350,016 as malware."
- TRANSFORMATION FUNCTION T(): Byte/Entropy histogram features, PE import features, String 2D histogram features, PE metadata features
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Threat score using Bayes' rule.

2016

Method #24. (L. Xu, 2016): In 2016, this paper introduced a malware detector that used static and dynamic analysis features for a neural network and SVM machine learning model. Xu reported an accuracy of 94.7%.

- CATEGORY: Malware Detector – Dynamic Analysis
- EFFICACY: accuracy of 94.7%
- TRAINING DATA: "In our dataset, 4002 samples are categorized as benign applications and 1886 samples are categorized as malware… We collected 5888 applications from Google Play and VirusShare. To reveal malicious and benign applications, we submit our samples to the VirusTotal web service and inspect the output of 51 commercial Anti-Virus (AV) scanners. We label all applications as malicious that are detected by at least two of the scanners. The other applications are labeled as benign. We end up with 1886

malicious applications and 4002 benign applications. The malicious samples were mostly discovered in 2014, and they are categorized into 39 families by a commercial AV scanner named AVG"

- TRANSFORMATION FUNCTION T(): Linux "strace" dynamic data. Static features from the samples: "In total, 10 static and dynamic feature sets are extracted from our malicious and benign Android applications including requested permissions, permission request APIs, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, instruction sequences, and system call sequences. In total, we generate 16 feature vector sets and 4 graph sets."
- PRE-PROCESSING FUNCTION P(): Generates graphs and "histogram, signature, n-gram, and the Markov Chain representations"
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks and SVM
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #25. (Gibert & Bejar, 2016): In 2016, this paper introduced a malware classifier that used static analysis features for a convolutional neural network machine learning model. They reported an accuracy of 98.56%.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: accuracy of 98.56
- TRAINING DATA: "For this challenge, Microsoft provided a dataset of 21741 samples, with 10868 for training and the other 10873 for testing, being a dataset of almost half a terabyte uncompressed. Microsoft provided a set of malware samples representing 9 different malware families. Each malware sample had an Id, a 20 character hash value uniquely identifying the sample and a class, an integer representing one of the 9 malware family names to which the malware belong: (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.ACY, (9) Gatak."
- TRANSFORMATION FUNCTION T(): Extract bytes and x86 instructions to create gray scale image representations of the sample.
- PRE-PROCESSING FUNCTION P(): Not available

- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #26. (Wojnowicz, Chisholm, Wolff, et al., 2016): In 2016, this paper introduced a malware detector that used structural entropy and wavelet analysis of the sample as features to a logistic regression, wavelet-based classifier. They reported an accuracy of 99% for parasitic malware.

- CATEGORY: Malware Detector – Structural Entropy
- EFFICACY: accuracy of 99% for parasitic malware
- TRAINING DATA: "Data were 699,121 samples of Portable Executable (PE) files from a Cylance repository. Of these samples, 17,605 files (2.51%) were parasitic malware, and the remaining files were legitimate software. We randomly selected 80% of the dataset for training, and the remaining 20% were allocated to the test set."
- TRANSFORMATION FUNCTION T(): Wavelet decomposition of structural entropy
- PRE-PROCESSING FUNCTION P(): Calculation of Structured Entropic Change Score (SSECS)
- MACHINE LEARNING MODEL FUNCTION MLM(): Wavelet-based classifier (logistic regression)
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Logistic regression parameters
- POST PROCESSING FUNCTION PP(): Not available

Method #27. (Wojnowicz, Chisholm, & Wolff, 2016): In 2016, this paper introduced a malware detector that used structural entropy as features to a logistic regression, wavelet-based classifier. They reported an accuracy of 68.7%.

- CATEGORY: Malware Detector – Structural Entropy
- EFFICACY: accuracy of 68.7%
- TRAINING DATA: "Data were a set of n=39,968 portable executable files from a Cylance repository. 19,988 (50.01%) of these files were known to be malicious, and the remaining files were benign."

- TRANSFORMATION FUNCTION T(): Wavelet decomposition of structural entropy
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Wavelet-based classifier (logistic regression)
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #28. (Narra, Troia, Corrado, Austin, & Stamp, 2016): In 2016, after the thesis published by Nara (Narra, 2015), this paper introduced a malware classifier based on hidden Markov models and the k-means clustering machine learning model. They reported an AUC of 0.977.

- CATEGORY: Malware Classifier – HMM
- EFFICACY: AUC of 0.977
- TRAINING DATA: "Most of the malware datasets used for this research were obtained from the Malicia project website. The Malicia dataset consists of more than 11,000 malware binaries collected from 500 drive-by download servers over a period of 11 months. All of the malware samples are in the form of executable (.exe) or dynamic-link library (.dll)files. Along with the malware samples, Malicia includes a database of metadata. Themetadata gives information as to when and where the malware was collected and, in most cases, the classification of the malware. Of the 11,000 malware samples in the dataset, about 7,800 were distributed among three dominant families, namely, Winwebsec, Zbot, and Zeroaccess. Our analysis is primarily focused on these three dominant datasets, but we do consider four additional datasets in some of our experiments."
- TRANSFORMATION FUNCTION T(): "For the research presented in this paper, we train HMMs for different compilers and malware families, based on extracted opcode sequences."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): k-means and expectation maximization

- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #29. (Kolosnjaji, Zarras, Webster, & Eckert, 2016): In 2016, this paper introduced a malware classifier that used dynamic analysis features for a convolutional and recurrent neural network machine learning model. They reported an accuracy of 89.4%.

- CATEGORY: Malware Classifier – Dynamic Analysis
- EFFICACY: accuracy of 89.4%
- TRAINING DATA: "We collect malware samples and trace malware behavior using a malware zoo. Our malware collection consists of samples gathered from three primary sources: Virus Share, Maltrieve and private collections. We select these sources to provide a large and diverse volume of samples for evaluation."
- TRANSFORMATION FUNCTION T(): "We choose the Cuckoo sandbox which is open source, widely-used, and provides a controlled environment for executing malware. During the execution of malware samples we record calls to the kernel API that we later use to characterize these malware samples. For each malware sample we obtain a sequence of API calls (system calls) and employ this sequence for behavioral modeling."
- PRE-PROCESSING FUNCTION P(): Redundant feature removal
- MACHINE LEARNING MODEL FUNCTION MLM(): "We construct a neural network based on convolutional and recurrent network layers in order to obtain the best features for classification. This way we get a hierarchical feature extraction architecture that combines convolution of n-grams with full sequential modeling."
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #30. (Gharacheh, Derhami, Hashemi, & Fard, 2016): In 2016, this paper introduced a malware detector that used static analysis and opcodes as features to a hidden Markov model machine learning model. They reported perfect accuracy.

- CATEGORY: Malware Detector – Static Analysis and Opcodes
- EFFICACY: perfect accuracy

- TRAINING DATA: "The proposed method is examined and tested on the well-known malware MWOR and the malware constructed by kits NGVCK, G2, MPCGEN, MetaPHOR and the random benign executable files which are selected from the Clang, Cygwin, GCC, MingW, TASM, Turbo C. The total number of malware and benign files in the training and test sets together is 1245."
- TRANSFORMATION FUNCTION T(): Extracts opcode sequences
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Hidden Markov models
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #31. (W. Huang & Stokes, 2016): In 2016, this paper introduced a malware classifier that used dynamic analysis features from a malware sample for a neural network. They reported an error rate of 2.94%.

- CATEGORY: Malware Classifier – Dynamic Analysis
- EFFICACY: error rate of 2.94%
- TRAINING DATA: "We were provided a large corpus of labeled, raw data by analysts from the Microsoft Corporation which was extracted from 6.5 million files. We believe this is the largest dataset used in a published malware classification study. Among this data collection, 2.85 million examples were extracted from malicious files and 3.65 million from benign files. The set of malicious files contained 1.3 million belonging to the 98 malware families and 1.55 million from the generic malware class. We randomly selected 4.5 million examples for training and 2.0 million for a hold out test set."
- TRANSFORMATION FUNCTION T(): "All models are trained with data extracted from dynamic analysis of malicious and benign files… For each executable file which is emulated by the anti-malware engine, two sets of raw information are extracted: a sequence of application programming interface (API) call events plus their parameters and a sequence of null-terminated objects recovered from system memory during emulation… The combined feature set consisting of null-terminated tokens, API event plus parameter value, and API trigrams contains millions of potential features… The

output of the feature selection process is a ranked set of 50,000 features which is input to the MtNet system. The 50,000 features are initially selected during training. Later, these features are applied when evaluating an unknown file"

- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #32. (L. Xu, Cavazos, Jayasena, & Cavazos, 2016): In 2016, this paper introduced a malware detector that used dynamic and static analysis features from the malware for the neural network and restricted Boltzmann machine. They reported an accuracy of 94.7%.

- CATEGORY: Malware Detector – Static and Dynamic Analysis
- EFFICACY: accuracy of 94.7%
- TRAINING DATA: "We collected 5888 applications from Google Play and VirusShare. To reveal malicious and benign applications, we submit our samples to the VirusTotal web service and inspect the output of 51 commercial Anti-Virus (AV) scanners. We label all applications as malicious that are detected by at least two of the scanners. The other applications are labeled as benign. We end up with 1886 malicious applications and 4002 benign applications. The malicious samples were mostly discovered in 2014, and they are categorized into 39 families by a commercial AV scanner named AVG."
- TRANSFORMATION FUNCTION T(): "We first extract static and dynamic information, and convert this information into vector-based representations." Requested permissions, permission request API, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, instruction sequences, system call sequences.
- PRE-PROCESSING FUNCTION P(): Generates graphs
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks and restricted Boltzmann machines
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available

- POST PROCESSING FUNCTION PP(): Not available

Method #33. (Wang et al., 2016): In 2016, this paper introduced a malware classifier that used static analysis features from the malware for the neural network. They reported an accuracy of 96%.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: accuracy of 96%
- TRAINING DATA: "We collected 5000 real-world malware samples from malware sharing sources like Vxvault, VirusTotal, Malshare and some other public resources. We scanned the samples with different AVs and kept only those samples where majority of AV detection name was same. So we kept only 17 different prevalent malware families from the initial sample set."
- TRANSFORMATION FUNCTION T(): "In this paper we proposed, implemented and evaluated a method, called ByteFreq that can cluster large number of samples using byte frequency. Byte frequency is represented as time series and SAX (Symbolic Aggregation approXimation) is used to convert the time series in symbolic representation."
- PRE-PROCESSING FUNCTION P(): Symbolic representations and clusters strings
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural network
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

2017

Method #34. (Yue, 2017): In 2017, Yue introduced a malware classifier that used gray scale images of the malware samples as the feature to convolutional neural networks. Yue reported an accuracy of 98.63%.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: accuracy of 98.63%
- TRAINING DATA: "For instance, the dataset used in our paper contains 25 classes, and some class contains more than 2000 images while some other has only 80 images or so."
- TRANSFORMATION FUNCTION T(): Converts byte code into gray scale images.

- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional Neural Networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): A weighted softmax loss parameter.
- POST PROCESSING FUNCTION PP(): Not available

Method #35. (Kolosnjaji, Eraisha, Webster, Zarras, & Eckert, 2017): In 2017, this paper introduced a malware classifier that used static analysis features to convolutional neural networks. They report an accuracy of 93%.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: accuracy of 93%
- TRAINING DATA: "We collected a set of malware samples over multiple months from three primary sources: Virusshare, Maltrieve and private collections… Our final dataset after noise filtering contains 22,757 executables, with 22,694 malicious executables and 63 benign executables from ZDNet's download list of most popular programs"
- TRANSFORMATION FUNCTION T(): Static features from the sample and opcode information. PE Metadata, PE Import features, Assembly Opcode features
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks and artificial neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #36. (Zhao, Han, & Meng, 2017): In 2017, this paper introduced a malware classifier that used an intermediate language representation of the opcodes for a long short-term memory (LSTM) neural network. They report an accuracy of 97.7% and AUC of 0.9832.

- CATEGORY: Malware Classifier – Intermediate Language of Opcodes
- EFFICACY: accuracy of 97.7% and AUC of 0.9832

- TRAINING DATA: "The malware dataset used in our experiment is obtained from VxHeaven, which is a well-known malware repository. Our dataset contains 310,000 malwares, with 210,000 labeled malwares and 100,000 unlabeled malwares."
- TRANSFORMATION FUNCTION T(): "MDIL leverages REIL (Reverse Engineering Intermediate Language) [8] as the intermediate representation and expands it to provide an abstract representation of the malware."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Long short term memory (LSTM)
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #37. (Yuxin & Siyi, 2017): In 2017, this paper introduced a malware detector based on opcode statistical features for a deep belief network (DBN). They report an accuracy of 98%.

- CATEGORY: Malware Detector – Opcode Statistics
- EFFICACY: accuracy of 98%
- TRAINING DATA: "In Table 1, the benign samples in each dataset are sampled from a big dataset containing 4600 benign files, and the malicious samples are sampled from a malware dataset containing 4600 samples… All the experimental samples are PE format files. The malicious samples are downloaded from the following Web sites: vx.netlux.org, www.kaggle.com/c/malware- classification/ and www.offensivecomputing, and the benign files come from the Windows system files."
- TRANSFORMATION FUNCTION T(): Sample is disassembled and opcode n-grams are used as features.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Deep believe networks (DBN)
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #38. (Kilgallon, De La Rosa, Cavazos, Rosa, & Cavazos, 2017): In 2017, this paper introduced a malware classifier that used static and dynamic features from a sample for a variety of machine learning model algorithms. They report an accuracy of 92%.

- CATEGORY: Malware Classifier – Static and Dynamic Analysis
- EFFICACY: accuracy of 92%.
- TRAINING DATA: "In this section, we present and discuss the major results from running and analyzing a dataset of 3320 malware samples using Cuckoo, and the additional implementation of the machine learning classifiers to predict the malware analysis run time, as well as predicting what family a malware belongs to."
- TRANSFORMATION FUNCTION T(): "Furthermore, we extract features from dynamic analysis performed in Cuckoo, and combine these features with static analysis features from r2 to form a hybrid feature vector that is used to create a malware family classification model… Our static features which are derived from static analysis are contextual byte features, portable executable (PE) import features, a 2-dimensional histogram of string features, and PE metadata features. Contextual byte features take the form of a byte-entropy histogram, which models the files distribution of bytes. From the PE header, we extract the PE Import and Metadata tables in which we hash each into a size 256 vector. Also, we extract all the strings in the file to be hashed into a vector as well. These features are combined to form a 1024 length vector, which will provide us with static information for that sample. Saxe and Berlin show that this set of static features can be used to build highly accurate machine learning models… In our dynamic analysis, we run the malware in a Cuckoo sandbox and extract its behaviors in JSON reports. We then convert these reports to dynamic features. The dynamic features that are used to build our machine learning models correspond to API calls from all malware seen in the dataset which are generated from dynamic analysis. In addition, each of the malware's dynamic features is comprised of an API call histogram."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Five particular classifiers are selected for our experiments: Decision Tree, Random Forest, Support Vector Machine, Neural Network, and K-nearest Neighbors. Finally, we use 10-fold cross validation to perform the training and testing of our algorithms."

- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #39. (Lui & Wang, 2017): In 2017, this paper introduced a malware detector that used gray scale images and opcode n-grams for a variety of machine learning model algorithms. They reported an accuracy of 98.5%.

- CATEGORY: Malware Detector – Images from Bytes and n-grams
- EFFICACY: accuracy of 98.5%
- TRAINING DATA: "Dataset: In the last two weeks, we capture the malware using Kingsoft, ESET NOD32, and Anubis from our campus network. The malware contains 4,755 samples, which are from 10 malware families. Benign software is collected under 32-bit windows 7. It includes 2,100 samples from the clients of campus."
- TRANSFORMATION FUNCTION T(): "We adopt gray scale image and OpCode n-gram to obtain high quality features."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Algorithms: seven classical algorithms are from the sckit-learn of python 2.7, such as Random Forest (RF), SVM (kernel = 'poly'), Logistic Regression (Logistic), Gradient Boosting Classifier (Boosting), GaussianNB (GNB), Decision Tree Regressor (DT) and Multilayer perceptrons (MLP). It should be noted that MLP is the deep learning structure mentioned earlier."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #40. (Vanderbruggen & Cavazos, 2017): In 2017, this paper introduced a malware classifier that used static analysis features for neural network and random forest. They reported an accuracy of 7.2% error rate.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: 7.2% error rate

- TRAINING DATA: "We obtained millions of files from Reversing Labs' stream of malware targeting financial institutions. These malware are from forty families, targeting a variety of systems. For the experiments presented in this paper, we randomly subsampled each family with more than one thousand variants. This left us with a dataset made of one thousand files from eleven different families."
- TRANSFORMATION FUNCTION T(): "We consider two categories of static analyses: bytes-level and code-level. Bytes-level analysis considers the raw bytes in the file. This includes: ASCII strings extraction [8], bytes N-grams [9], [10], and statistical measures (i.e. entropy) [11]. One of the most comprehensive bytes analysis is the extraction of the bytes-entropy histogram of a file [12]. This analysis identifies the amount of information associated with various bytes distributions providing a highly representative image of the file. Code-level analyses involves the domain of reverse-engineering. It considers the executable portion of the files, x86 instructions, byte-code instructions, or scripting languages. Tools like IDApro [13] or Radare2 [14] can extract executable code from various type of files"
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks, Random forests
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #41. (Rezende, Ruppert, Carvalho, Ramos, & Geus, 2017): In 2017, this paper introduced a malware classifier that used images made from the malware sample's byte values as features with a convolutional neural network. They reported an accuracy of 98.62%.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: accuracy of 98.62%
- TRAINING DATA: "The DCNN method has been evaluated with the Malimg dataset proposed by Nataraj et al. The dataset comprises 9,339 byteplot images from 25 different malware families. As reported by the authors, all byteplot images were generated from

malware executables submitted to the Anubis analysis system and labels provided by Microsoft Security Essentials were used to obtain the ground truth for the dataset."

- TRANSFORMATION FUNCTION T(): "Malware samples are represented as byteplot grayscale images and a deep neural network is trained freezing the convolutional layers of ResNet-50 pre-trained on the ImageNet dataset and adapting the last layer to malware family classification"
- PRE-PROCESSING FUNCTION P(): "Convert each byteplot to an RGB image, rescaling it to 224 × 224 dimension and subtracting the mean RGB value computed on the ImageNet dataset from each pixel, as proposed by Krizhevsky et al., to feed it to the deep neural network"
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks and ResNet-50
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #42. (Shijo, P.V. Salim, 2017): In 2017, this paper introduced a malware detector using static and dynamic analysis of the malware sample for a variety of machine learning model algorithms. They reported an accuracy of 98.62%.

- CATEGORY: Malware Detector – Static and Dynamic Analysis
- EFFICACY: accuracy of 98.62%
- TRAINING DATA: "For malware, we approached VirusTotal, which is one of the biggest platforms where we can analyze any malware sample, and they allowed us to use their paid VirusTotal Intelligence Service for 2 months. We collected around 15,000 malware samples from their platform. Final working data set include 11347 malware as some of the samples either could not be parsed by pefile Python module or some of them could not be run inside Virtual Machine due to some errors… For benign executables, we used a simple python script to scrape all the valid executables from a freshly installed windows 7 and windows XP on a virtual machine. We were able to collect around 773 files. 1000 Benign fileswere downloaded from the VirusTotal intelligence search by restricting the number of positives for all the anti-virus vendors to be zero."

- TRANSFORMATION FUNCTION T(): Strings, Pefile parser, Cuckoo Sandbox
- PRE-PROCESSING FUNCTION P(): COFF file and optional header attributes; histograms related to strings length; entropy measures using sliding window on bytes, file system related features; mutexes names and related features; registry changes related features
- MACHINE LEARNING MODEL FUNCTION MLM(): KNN, Random Forest, Decision Tree, Xg-Boost Classifiers
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #43. (Mosli, Li, Yuan, & Pan, 2017): In 2017, this paper introduced a malware detector using dynamic analysis of the malware sample with a variety of machine learning model algorithms. They reported an accuracy of 91.4%.

- CATEGORY: Malware Detector – Dynamic Analysis
- EFFICACY: accuracy of 91.4%
- TRAINING DATA: "The dataset comprised 3,130 malware samples from the VirusShare malware repository [28]. Additionally, 1,157 benign software samples were collected from various locations such as the Windows System32 directory and from software websites such as FileHippo."
- TRANSFORMATION FUNCTION T(): "After dynamically obtaining the handle data by running the software in a sandbox, machine learning is used to discriminate between benign and malicious uses of handles and to generalize handle usage behavior to detect previously-unknown malware… The four Windows virtual machines were run concurrently, each with an instance of benign or malicious software. During the analysis task, a memory dump was produced of each Windows virtual machine along with a report with content and behavioral information about the sample. Furthermore, VirusTotal was used to scan each sample to ensure that the sample was labeled correctly as benign or malicious, and then determine the malware family to which it belonged… Handle data was extracted from the memory dumps of machines with benignware or malware using Volatility."

- PRE-PROCESSING FUNCTION P(): "The term frequency-inverse document frequency (TF-IDF) was used to extract measurable features from the Volatility handles out- put; the extraction and model training was implemented using scikit-learn. A vocabulary was created comprising the handle types to be extracted from the handle text files. A list of all the possible terms in the vocabulary was obtained from Schuster."
- MACHINE LEARNING MODEL FUNCTION MLM(): "Three machine learning models were compared: (i) k-nearest neighbor (KNN); (ii) support vector machine (SVM); and (iii) random forest."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #44. (Pai, Troia, Visaggio, Austin, & Stamp, 2017): In 2017, this paper introduced a malware classifier that used hidden Markov models and k-means clustering. They reported an accuracy of greater than 90%.

- CATEGORY: Malware Classifier – HMM
- EFFICACY: accuracy of greater than 90%
- TRAINING DATA: "Most of our malware samples were obtained from the Malicia project website; see also [19]. The Malicia dataset consists of about 11,000 malware binaries. However, for our experiments we focus primarily on the three dominant families in the dataset, namely, Zbot, ZeroAccess, and Win-websec."
- TRANSFORMATION FUNCTION T(): Hidden Markov models
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): k-means and expectation maximization
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Number of clusters and dimensions
- POST PROCESSING FUNCTION PP(): Not available

Method #45. (Choi, Jang, Kim, & Kim, 2017): In 2017, this paper introduced a malware classifier that used gray scale images of the malware sample's bytes as features for a convolutional neural network. They reported an accuracy of 95.66%.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: accuracy of 95.66%
- TRAINING DATA: 10,000 benign and 2,000 malware files from vendors.
- TRANSFORMATION FUNCTION T(): Generates images from the byte code from samples.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks.
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #46. (Han, 2017): In 2017, this thesis introduced a malware trojan detector using dynamic analysis of the malware sample for a variety of machine learning model algorithms. They reported an accuracy of up to 98.3%.

- CATEGORY: Malware Detector – Dynamic Analysis
- EFFICACY: accuracy up to 98.3%
- TRAINING DATA: "The experiment dataset includes 3000 Portable Executable samples, among which 1500 are malicious and the other 1500 are benign software. The ground truth labels are obtained through Virus Total online scanner. We divide the sample set into three subsets with 1000 samples each. The first subset is for initial training, the second subset is for initial testing and continuous updating of the classifiers, and the third subset is for continuous testing."
- TRANSFORMATION FUNCTION T(): "In our current implementation we considered two options for the training phase. The first option relies on manual exploration, where users exercise most of the core functionality of each application in the training set. While this does not technically ensure 100% coverage of code paths, it sufficiently captures the code paths commonly executed by normal users. The second option relies on symbolic execution to improve the code coverage and thus the functionality coverage. We will explore this option in future work. The full results will be delivered in this thesis."
- PRE-PROCESSING FUNCTION P(): Not available

- MACHINE LEARNING MODEL FUNCTION MLM(): "TroGuard built 5 models using the meta classifier Ensembles of Nested Dichotomies (END) in the Weka machine learning suite, based on the C4.5 decision tree classification algorithm."
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #47. (Rhode, Burnap, & Jones, 2017): In 2017, this paper introduced a malware detector that used dynamic analysis of the sample for a recurrent neural network. They report an accuracy of 94%.

- CATEGORY: Malware Detector – Dynamic Analysis
- EFFICACY: accuracy of 94%.
- TRAINING DATA: "We initially obtained 1,000 malicious and 600 "trusted" Windows7 executables from VirusTotal along with 800 trusted samples from the system files of a fresh Windows7 64- bit installation. We then downloaded a further 4,000 Windows 7 applications from popular free software sources, such as Softonic, PortableApps and SourceForge. We included the online download files as they are a better representation the typical workload of an anti-virus system than Windows system files… The final dataset comprised 2,345 benign and 2,286 malicious samples, which is consistent with dataset sizes in this field of research e.g. ([omitted]). We used a further 2,876 ransomware samples obtained from the VirusShare online malware repository for the ransomware case study"
- TRANSFORMATION FUNCTION T(): "In this paper we investigate the possibility of predicting whether or not an executable is malicious based on a short snapshot of behavioural data." Collected were: total processes, max process ID, CPU User, CPU System, Memory Use, Swap Use, Packets sent, Packets received, Bytes Received, Bytes Sent
- PRE-PROCESSING FUNCTION P(): "Prior to training and classification, we normalise the data to improve model convergence speed in training. By keeping data between 1 and -1, the model is able to converge more quickly, as the neurons within the network operate

within this numeric range. We achieve this by normalising around the zero mean and unit variance of the training data."

- MACHINE LEARNING MODEL FUNCTION MLM(): Recurrent neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): "We conduct a random search of hyperparameter configurations and provide details of the configurations leading to high classification accuracy, giving insight into the methods required for optimising our malware detection model."
- POST PROCESSING FUNCTION PP(): Not available

Method #48. (Raff et al., 2017): In 2017, this paper introduced a malware detector that used static analysis features from the sample for a recurrent neural network. They reported an accuracy of 94% and an AUC of 98.1.

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy of 94% and AUC of 98.1
- TRAINING DATA: "For this work we use the same training and testing data as in Raff et al. (2016). Specifically, we use the Group B training data, and Group A & B testing data. Group B data was provided by an anti-virus industry partner, where both the benign and malicious programs are meant to be representative of files seen on real machines. The Group B training set consists of 400,000 files split evenly between benign and malicious classes. The testing set has 77,349 files, of which 40,000 are malicious and the remainder are benign. The Group A data was collected in the same manner as most work in the malware identification literature (Schultz et al. 2001; Kolter and Maloof 2006) , which is available to the public. The benign data (or "goodware") comes from a clean installation of MicrosoftWindows, with some commonly installed applications (e.g., Firefox, Flash, etc) and the malware comes from the VirusShare corpus (Roberts 2011). The Group A test set contains 43,967 malicious and 21,854 benign testing files… In addition, reaching out to the original company, we have obtained a larger training corpus of 2,011,786 binaries, with 1,000,020 benign and 1,011,766 malicious. We use this larger dataset to show that our new MalConv architecture continues to improve with increased training data, while the byte n-gram approach appears to have plateaued in terms of performance."

- TRANSFORMATION FUNCTION T(): "In this work we introduce malware detection from raw byte sequences as a fruitful research area to the larger machine learning community… We instead take a static analysis approach, where we look at information from the binary program that can be obtained without running it. In particular, we look at the raw bytes of the file itself, and build a neural network to determine maliciousness."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Recurrent neural networks
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #49. (Z. Xu, Wen, Qin, & Ming, 2017): In 2017, this paper introduced a malware detector that used static analysis and opcode features for a variety of machine learning model algorithms. They reported an accuracy of 97.88% and an AUC of 0.9959.

- CATEGORY: Malware Detector – Static Analysis and Opcodes
- EFFICACY: accuracy of 97.88 and AUC of 0.9959
- TRAINING DATA: "Our dataset consists of a malware dataset and a benign software dataset. The malware dataset consists of the samples of the BIG 2015 Challenge and the samples before 2017 in theZoo aka Malware DB, with 11376 samples in total; while the benign software dataset are collected from QIHU 360 software company, which is the biggest Internet security company in China, with 8003 samples in total."
- TRANSFORMATION FUNCTION T(): "Different from most existing work, we take into account not only the behaviour information but also the data information, namely, the opcodes, data types and system libraries used in executables… This section presents the processing of the feature extractor, which consists of the following steps: (1) decompilation, (2) information extraction, and (3) feature selection and representation."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Moreover, in our implementation, we employ various machine learning methods, such as K-Nearest Neighbor, Native Bayes, Decision Tree, Random Forest, and Support Vector Machine to train our classifier"

- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #50. (Meng et al., 2017): In 2017, this paper introduced a malware classifier that used static analysis and opcode features for a convolutional neural network. They reported an accuracy of 98%.

- CATEGORY: Malware Classifier – Static Analysis and Opcodes
- EFFICACY: accuracy of 98%
- TRAINING DATA: "We choose the data-sets from VX Heavens for our experiment, which is an important malware database for malware research. The data-sets including Trojan and Backdoor consist of 5647 malwares from four families."
- TRANSFORMATION FUNCTION T(): "In this section, we use word2vec method to make a distributed representation for each malware gene. The basic idea is that each malware gene is trained to a k-dimensional real vector, thus all the malware gene sequences were converted into an n×k two-dimensional matrix, where n is the length of the malware gene sequences. Each vector represents a point in the k-dimensional space, and each element of the vector is determined by repeatedly training and adjusting the weight for the gene sequences. The spatial correlation between the vectors is used to represent the semantic relevance and similarity of the malware genes. In the paper, we consider API calls as the malware genes and express intrinsic correlation of API calls using word2vec method. API calls are functional, such as openfile(), writefile(), deletefile() etc.. All of these are API calls for file operations, which are semantically similar. However, most existing methods of matching similar API calls are artificial. With word2vec method, all the API calls are expressed effectively and the semantic similarity is expressed automatically."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural network
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #51. (Athiwaratkun & Stokes, 2017): In 2017, this paper introduced a malware classifier that used dynamic analysis features for a long short-term memory (LSTM) model and a gated recurrent unit (GRU) model. They reported a false positive rate of 1% and a 31.3% improvement in the true positive rate of compared methods.

- CATEGORY: Malware Classifier – Dynamic Analysis
- EFFICACY: false positive rate of 1% and a 31.3% improvement in the true positive rate of compared methods
- TRAINING DATA: "Before presenting the proposed models in the next section, we first describe the dataset used in our work. A modified version of the production Microsoft anti-malware engine was used to perform dynamic analysis of 75,000 Windows portable executable (PE) format files equally split between malware and benign files. Our dataset is further randomly split into three datasets including a training set with 50,000 files, a validation set with 10,000 files, and a test set with 15,000 files"
- TRANSFORMATION FUNCTION T(): Samples are detonated and system calls are recorded.
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "In this work, we propose several new malware classification architectures which include a long short-term memory (LSTM) language model and a gated recurrent unit (GRU) language model… Finally, we propose a new single-stage malware classifier based on a character-level convolutional neural network (CNN)."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #52. (T. H. Huang & Kao, 2017): In 2017, this paper introduced an Android malware detector that used static analysis features of the sample for a convolutional neural network. They reported an accuracy of more than 75%.

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy of more than 75%.

- TRAINING DATA: "We have collected over 1 million malware samples and 1 million benign samples from the data provided by our research cooperate partner's (Leopard Mobile Inc.) core products Security Master and Clean Master (which had reached 3,810 million installations globally with 623 million monthly active users by December 2016)… The data was collected from January to May in 2017, among which we had a collection of approximately 1.5 million of benign and malicious Android apps for our experiments. We keep our research results and data on the website http://R2D2.TWMAN.ORG (72 GB). So far, we have accumulated 2 million effective data collections and continue to train and adjust our model."
- TRANSFORMATION FUNCTION T(): "In particular, we develop a color representation for translating Android apps into RGB color code and transform them to a fixed-sized encoded image… Since the byte-code in classes.dex and rgb color code are both hexadecimal, we proposed a transformation algorithm that enables the bytecode as rgb color code and into color image (called Android color images)."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Also, based on the successful study in image detection in CNN, and the method proposed by [21]. Thus, we attempt to find out the hidden relationship between the program execution logic and the order of function calls behind the malware by taking advantage of the CNN in order to accurately detect known and even unknown malware… Hence we present a coloR-inspired convolutional neuRal networks (CNN)-based AndroiD malware Detection (R2-D2), which can detect malware without extracting pre-selected features (e.g., the control-flow of op-code, classes, methods of functions and the timing they are invoked etc.) from Android apps… After that, the encoded image is fed to convolutional neural network for automatic feature extraction and learning, reducing the expert's intervention… After that, we apply CNN to the Android color images for the Android malware detection."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #53. (McLaughlin et al., 2017): In 2017, this paper introduced an Android malware detector that used static analysis and opcode features from the sample for a convolutional neural network. They reported an accuracy of 98%.

- CATEGORY: Malware Detector – Static Analysis and Opcodes
- EFFICACY: accuracy of 98%
- TRAINING DATA: "The second dataset was provided by McAfee Labs (now Intel Security) and comes from the vendor's internal repository of Android malware. After discarding empty files or files that are less than 8 opcodes long, the dataset contains 2475 malware samples and 3627 benign samples. This dataset does not include malware family labels and may include malware and/or benign applications present in the small dataset. Hence to ensuring training hygiene i.e. to ensure we do not train on the testing-set, the network is trained and tested on each dataset separately without cross- contamination. We refer to this dataset as the 'Large Dataset'. We also have an additional dataset provided by McAfee Labs containing approximately 18,000 android programs, and which was collected more recently than the first two datasets. This was used for testing the final system after setting the hyper-parameters using the smaller datasets. After discarding short files, the dataset contains 9268 benign files and 9902 malware files. We refer to this dataset as the 'V. Large Dataset'."
- TRANSFORMATION FUNCTION T(): "Malware classification is performed based on static analysis of the raw opcode sequence from a disassembled program… The training pipeline of our proposed system is much simpler than existing n-gram based malware detection methods, as the network is trained end-to-end to jointly learn appropriate features and to perform classification, thus removing the need to explicitly enumerate millions of n-grams during training… In this work we propose a malware detection method that uses a convolutional network to process the raw Dalvik byte-code of an Android application."
- PRE-PROCESSING FUNCTION P(): Opcode embedding
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #54. (Tran & Sato, 2017): In 2017, this paper introduced a malware classifier that used dynamic analysis features for various machine learning model algorithms. They reported an accuracy of 99.06%.

- CATEGORY: Malware Classifier – Dynamic Analysis
- EFFICACY: accuracy of 99.06%
- TRAINING DATA: "For the experiments, we make use of the dataset implemented and shared by Ki Y. This dataset was built from 23,080 malware samples randomly from Malicia project and VirusTotal, and it was shared online (via URL http://ocslab.hksecurity.net/apimds-dataset)."
- TRANSFORMATION FUNCTION T(): "The proposed malware classification procedures introduced in this paper use? API call sequences as inputs to classifiers. In addition, taking advantage of the development in Natural Language Processing field, we use some methods such as n-gram, doc2vec (or Paragraph vectors), TF-IDF to convert those API sequences to numeric vectors before feeding to the classifiers. Our proposed approaches are divided into 3 different methods to classify malware, that is TF-IDF, Paragraph Vector with Distributed Bag of Words and Paragraph Vector with Distributed Memory."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): SVM, k-nearest neighbor, MLP, random forests
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #55. (Wojnowicz et al., 2017): In 2017, this paper introduced a malware detector that used structural entropy as features for a logistic lasso regression machine learning model. They reported an accuracy of 96.62%. Adding additional features brought the accuracy up to 98.62%.

- CATEGORY: Malware Detector – Structural Entropy
- EFFICACY: accuracy of 96.62% and 98.62%, depending on the feature selection

- TRAINING DATA: "Samples were a corpus of Portable Executable (PE) files downloaded from Virus Total in June, 2014. Files were labeled as "clean" if no trusted Anti-Virus vendors labeled the file as malware, whereas files were labeled as "malware" if one or more trusted Anti-Virus vendors labeled the file as malware. We formed a balanced data set of 39,969 files; of these, 19,991 (50.01%) were labeled as malware, whereas 19,978 (49.98%) were labeled as clean computer programs. The "malware" category contained different types of malicious software (e.g., computer viruses, Trojan horses and spyware –but not adware). Before feature extraction, 31,960 (80%) of the samples were randomly selected to belong to the "training set", and the other 7991 (20%) of the samples were classified in the "test set". The training set was used not only (a) to train the feature weights in the final classifier, but also (b) to determine the sparsity of the final classifier and (c) to determine weights for constructing the "propensity features" described below."
- TRANSFORMATION FUNCTION T(): "In particular, SUSPEND (a) quantifies the "amount of structure" in the entropy signal (through detrended fluctuation analysis), (b) finds the location and size of sudden jumps in entropy (through mean change point modeling), and (c) computes the distribution of entropic variation across multiple spatial scales (through wavelet decomposition). In addition, SUSPEND (d) summarizes the entropy signal's empirical probability distribution." The structural entropy is calculated.
- PRE-PROCESSING FUNCTION P(): Detrended Fluctuation Analysis (DFA), Mean Changepoint Modeling (MCPM), Wavelet Decomposition, Summary Statistics
- MACHINE LEARNING MODEL FUNCTION MLM(): "In principle, any ML classifier could be used by SUSPEND to relate the features extracted from the entropy signal to malware status. In this paper, we apply a logistic lasso regression, as the interpretability of this model is particularly useful for investigating performance."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Calculates entropic suspiciousness score

Method #56. (Yakura, Shinozaki, Nishimura, Oyama, & Sakuma, 2017): In 2017, this paper introduced a malware classifier that used gray scale images of the malware's bytes as features for a convolutional neural network. They reported a 31.34% error.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: 31.34% error
- TRAINING DATA: "Dataset In the experiment, VX Heaven malware dataset was used because it has numerous samples and because it is used in many research efforts including [22]. As the ground-truth information of which malware family to which each sample belongs, detection results by Microsoft Security Essentials was used in the manner described in [22]. By excluding families with a small population (the threshold was set to 60), we obtained 147,803 samples from 542 families."
- TRANSFORMATION FUNCTION T(): Converts byte code to an image
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "This method, by applying convolutional neural network (CNN) with a technique called attention mechanism to an image converted from binary data, enables calculation of an "attention map," which shows regions having higher importance for classification in the image."
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

2018

Method #57. (Cakir & Dogdu, 2018): In 2018, this paper introduced a malware classifier that used static analysis features for a gradient boosting algorithm. They reported an accuracy of 95%.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: accuracy of 95%
- TRAINING DATA: "Microsoft released a large malware dataset in 2015 WWW Conference. It is called Microsoft Malware Classification Challenge Dataset (BIG 2015). We used this dataset in this work. Training dataset has 10869 sample malware from 9 different classes of malware. These malware classes are (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.ACY, (9) Gatak. The data distribution among these malware classes are shown in Figure 1. We

sampled 100, 200, 300, and 398 files from all classes (except class 5) and used these samples in order to show the performance of generated classification models trained with these low number of samples. We excluded class 5 (Simda) samples since it has a very low number of samples (42) and we wanted an even number of samples from each class in the experiments to remove the bias."

- TRANSFORMATION FUNCTION T(): "In this paper a shallow deep learning-based feature extraction method (word2vec) is used for representing any given malware based on its opcodes."
- PRE-PROCESSING FUNCTION P(): "Word2Vec is a popular tool used in the literature especially for natural languages like English. In this model an association of each word is accomplished with a mathematical vector representation under which some structural or semantic relation holds. Despite being designed for natural language processing, it is also satisfactory for malware assembly code representation."
- MACHINE LEARNING MODEL FUNCTION MLM(): "Gradient Boosting algorithm is used for the classification task."
- ERROR FUNCTION E(): Confusion matrices and logarithmic loss
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #58. (Ni, Qian, & Zhang, 2018): In 2018, this paper introduced a malware classifier that used static analysis and gray scale image features for a convolutional neural network. They reported an accuracy of 98.862% on average and 99.260% at best.

- CATEGORY: Malware Classifier – Static Analysis and Images from Bytes
- EFFICACY: accuracy of 98.862% on average
- TRAINING DATA: "The experimental data in this paper is from the Malware Classification Challenge on Kaggle by Microsoft 2015. There are 10,868 labeled malware samples from 9 families in this dataset with binary and disassembly files. After preprocessing, 10,805 samples remain. 80% of them will be used for training and the rest for testing."
- TRANSFORMATION FUNCTION T(): "In this paper we propose a malware classification algorithm that uses static features called MCSC (Malware Classification

using SimHash and CNN) which converts the disassembled malware codes into gray images based on SimHash and then identifies their families by convolutional neural network."

- PRE-PROCESSING FUNCTION P(): Image generation from SimHash.
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #59. (Hassen, 2018): In 2018, this thesis introduced a malware classifier that used function call graphs from static analysis as features to a neural network. Hassen reported a 97.9% accuracy.

- CATEGORY: Malware Classifier – Static Analysis
- EFFICACY: accuracy of 97.9%
- TRAINING DATA: "For the purpose of evaluating our proposed approach for classifying malware into families, we will be using the Microsoft Malware Classification Challenge (BIG 2015) dataset. The original dataset consists of 10,867 labeled malware samples. Our disassembled file parser were able to properly parse 10,260 of the samples. Hence, we will be using these in the following evaluations. The class distribution of these samples are shown in Table 4.1. To compare our work with Adagoi, we will use a secondary dataset consisting of 1,113 benign android apps and 1,200 malicious android apps. The malicious samples are from the Android Malware Genome Project. A colleague at our university provided us with the benign samples, downloaded from the Google Play Store."
- TRANSFORMATION FUNCTION T(): "We propose a method for extracting features from function call graphs (FCGs)."
- PRE-PROCESSING FUNCTION P(): Function clustering.
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): This research proposes a classifier that can adapt to open set malware classification, which can include a tuning function for this task.

- POST PROCESSING FUNCTION PP(): Not available

Method #60. (Le, Boydell, Mac, & Scanlon, 2018): In 2018, this paper introduced a malware classifier that used sampled byte code of the malware sample for a convolutional and BiLSTM neural networks. They reported an accuracy of 98.2%.

- CATEGORY: Malware Classifier – Byte Code
- EFFICACY: accuracy of 98.2%
- TRAINING DATA: "For our experiments, we used the malware data from the Microsoft Malware Classification Challenge (BIG 2015) on Kaggle. Although the Kaggle challenge itself finished in 2015, the labeled training dataset of 10,868 samples is still available and represents a large collection of examples classified into malware classes, as shown in Table 1. As well as being able to use this data to both train and evaluate our own deep learning approaches, the Kaggle challenge still allows the submission of predictions for a separate unlabeled test set of 10,873 samples for evaluation."
- TRANSFORMATION FUNCTION T(): "Our 1 dimensional representation of a raw binary file is similar to the image representation of a raw binary file; but it is simpler, and it preserves the sequential order of the byte code in the binaries. The sequential representation makes it natural for us to apply the Convolutional Neural Network - Bi Long Short Term Memory architecture (CNN-BiLSTM) on top of it; helping us achieve better performance than using the CNN model alone… To this end, we used a generic image scaling algorithm, where the file byte code is interpreted as a one dimensional 'image' and is scaled to a fixed target size. This is a type of lossy data compression. However, by using an image scaling algorithm, we aim to limit the distortion of spatial patterns present in the data… In our experiments that follow, we scale each raw malware file to a size of 10,000 bytes using the OpenCV computer vision library [37] - i.e. after the scaling one malware sample corresponds to one sequence of 10, 000 1-byte values."
- PRE-PROCESSING FUNCTION P(): Sampling the byte code to fixed size
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks and BiLSTM neural networks
- ERROR FUNCTION E(): Not available
- TUNING FUNCTION TU(): Not available

- POST PROCESSING FUNCTION PP(): Not available

Method #61. (Yan, Qi, & Rao, 2018): In 2018, this paper introduced a malware classifier that used byte code images and opcode features for CNN and LSTM networks. They reported an accuracy greater than 99%.

- CATEGORY: Malware Classifier Images from Bytes and Opcodes
- EFFICACY: accuracy greater than 99%
- TRAINING DATA: "To verify the performance of MalNet, we perform evaluation experiments on a large dataset, which contains 21,736 malware samples from Microsoft and 20,650 benign samples collected by us. We choose 1/10 samples as validation dataset, where MalNet achieves detection accuracy of 99.88% and true positive rate of 99.14% with a false positive rate of 0.1%, much higher than the n-gram baseline result. Meanwhile we also make a malware family classification for 21,736 malware samples in 9 malware families to compare MalNet with other related works, where MalNet achieves 99.36% overall accuracy outperforming most of other methods. Besides, since rapid growing malware samples require a fast and efficient malware detection method, we evaluate the detection efficiency for MalNet."
- TRANSFORMATION FUNCTION T(): "In this paper, we propose MalNet, a novel malware detection method that learns features automatically from the raw data… MalNet performs a comprehensive static analysis which includes two novel methods based on deep neural networks. One method is learning from grayscale images by Convolution Neural Network (CNN). The grayscale image is extracted from raw binary file in which CNN can get the structure features of a malware from its local image patterns. The other method is learning from opcode sequence by Long-Short Term Memory (LSTM). Opcode sequences are extracted by decompilation tool where LSTM can learn features about malicious code sequences and patterns."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "Then MalNet uses CNN and LSTM networks to learn from grayscale image and opcode sequence, respectively, and takes a stacking ensemble for malware classification… Overall, MalNet uses these two networks to learn features from the raw data and then uses stacking ensemble to fuse two

networks' discriminant result with extra metadata feature and finally generates a binary classification result for malware detection."

- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #62. (Saleh, Li, & Xu, 2018): In 2018, this paper introduced a malware detector that used static and dynamic analysis features for a naïve Bayes and random forest classifier. They report an accuracy of 96.7%.

- CATEGORY: Malware Detector – Static and Dynamic Analysis
- EFFICACY: accuracy of 96.7%
- TRAINING DATA: "The programs were first seen by VirusTotal during the period from December 2014 to April 2015. We collected 100,000 malicious programs and 100,000 benign programs during each month between December 2014 and April 2015, which leads to a total of 1,000,000 programs."
- TRANSFORMATION FUNCTION T(): "In this paper, we combine three kinds of contextual information, namely static, dynamic, and instruction-based, for malware detection. This leads to the definition of more than thirty thousand features, which is a large features set that covers a wide range of a sample characteristics… First, we define and introduce a large set of features, including both static and dynamic aspects of malware samples. Specifically, we define more than 32,000 features. To the best of our knowledge, this is the largest feature set that has been defined for malware analysis… Therefore, we collect several features from many aspects of a malicious file, and use them to build our feature model for describing a system of malware. These feature sets include dynamic analysis of the file when it runs in a controlled environment, static analysis of the file header features, and data based on the instructions that compose the executable file… Combining these three sets culminates in a feature vector containing all the features. Our vector consists of 32,069 features… The scan report from VirusTotal is a JSON file containing static and dynamic analysis data of the file. The static analysis data is obtained using the pefile tool, while the dynamic analysis is done by a modified version of Cuckoo."
- PRE-PROCESSING FUNCTION P(): Not available

- MACHINE LEARNING MODEL FUNCTION MLM(): "Second, we use Random Forest and Naive Bayes machine learning techniques to learn malware detection models based on this large set of features… We use two popular machine learning algorithms to conduct our experiments: Naive Bayes and Random Forest."
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #63. (Krcal, Svec, & Balek, 2018): In 2018, this paper introduced a malware detector that used a variety of high- and low-level features for a neural network. They report a 98.55% accuracy.

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy of 98.55%
- TRAINING DATA: "Out of the Avast's repository of PE files we have chosen all those collected during recent 16 months of size between 12 and 512 kilobytes excluding files with some obfuscation methods such as compression or encryption detected. The train, validation and test sets consists of the first 12 months, the next 2 months and the last 2 months, respectively, so that we measure how the model generalizes into the future. For the sake of simplicity, we use binary labels clean and malware only with roughly balanced occurrence throughout our dataset."
- TRANSFORMATION FUNCTION T(): Samples are unpacked. "Our approach is the end-to-end counterpart of so-called static malware analysis: the network is given mere sequence of bytes that the executable consists of."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #64. (Yousefi-Azar, Hamey, Varadharajan, & Chen, 2018): In 2018, this paper introduced a malware detector that used a variety of high- and low-level features for a neural network. They report an accuracy of 99.45% on Windows PE files.

- CATEGORY: Malware Detector – Static Analysis
- EFFICACY: accuracy of 99.45%
- TRAINING DATA: Approximately 96,625 samples from a variety of sources.
- TRANSFORMATION FUNCTION T(): "We show feature extraction, which is performed by tf-simhashing, is equivalent to the first layer of a particular neural network… The byte representation contains important information such as APIs, op-codes. The model learns the pattern of bytes. The tf-simhashing static feature representation is a fast solution to embed the byte patterns into a short size vector. Malytics generalizes the patterns well even for novel samples. Also, it not easy to craft adversarial samples to attack Malytics… Locality Sensitive Hashing (LSH) is one of the main categories of hashing methods. It hashes input data so that similar data maps to the same "buckets" with high probability, maximizing the probability of a "collision" for similar inputs. Simhashing is one of the most widely used LSH algorithms, adopted to find similar strings.""
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): "The three phases are implemented by a neural network with two hidden layers and an output layer."
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #65. (Xiaofeng, Xiao, Fangshuo, Shengwei, & Jing, 2018): In 2018, this paper introduced a malware detector that used dynamic analysis of malware samples for a recurrent neural network model. They report an AUC of 0.993.

- CATEGORY: Malware Detector – Dynamic Analysis
- EFFICACY: AUC of 0.993
- TRAINING DATA: "The experiment samples come from four data sets: normal samples from windows7 and windows xp system exe files, the malicious samples were randomly picked up from the Virus Share, VirusTotal. A open source sandbox, cuckoo, is used to monitor the system's kernel API call during the sample execution. In our experiment, the sandbox executed every sample for two minutes. The labels of samples were derived

from VirusTotal.com after a weighted probability calculation to the detection results of VirusTotal.com."

- TRANSFORMATION FUNCTION T(): "This paper proposes a system architecture that combines LSTM deep learning model based on sequence data and machine learning model based on API statistical features."
- PRE-PROCESSING FUNCTION P(): Removal of redundant data.
- MACHINE LEARNING MODEL FUNCTION MLM(): "In this paper, recurrent neural network model is used to extracts the abstract features… We combined the LSTM deep learning model [2] and random forests model and proposed a malware detection architecture, ASSCA"
- ERROR FUNCTION E(): Confusion matrices and ROC curves
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #66. (Kalash et al., 2018): In 2018, this paper introduced a malware classifier that used gray scale images of the malware's bytes for a convolutional neural network. They report a 99.97% accuracy.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: accuracy of 99.97%
- TRAINING DATA: "Malimg Dataset: This dataset [5] has a total of 9,339 malware samples that are represented as grayscale images. Each malware sample in the dataset belongs to one of the 25 malware families/classes. Also, the number of samples belonging to a malware family vary across the dataset. In our experiments, we randomly select 90% of malware samples in a family for training and the remaining 10% for testing. At the end, we have 8,394 malware samples for training and 945 samples for testing… Microsoft Malware Dataset: In 2015, Microsoft hosted a Kaggle competition for malware classification. In this challenge, Microsoft released a huge dataset (almost half a terabyte when uncompressed) consisting of 21,741 malware samples. This dataset is divided in two parts, 10,868 samples for training and the other 10,873 samples for testing. Each malware sample belongs to one of 9 different malware families. Like the Malimg dataset, the distribution of malware samples over classes in the training data is

not uniform and the number of malware samples of some families significantly outnumbers the samples of other families."

- TRANSFORMATION FUNCTION T(): "We convert malware binaries to grayscale images and subsequently train a CNN for classification."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural networks
- ERROR FUNCTION E(): Confusion matrices and logloss
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #67. (Su et al., 2018): In 2018, this paper introduced a malware classifier that used gray scale images of the malware's bytes for a convolutional neural network. They reported an 81.8% classification accuracy.

- CATEGORY: Malware Classifier – Images from Bytes
- EFFICACY: accuracy of 81.8%
- TRAINING DATA: "For these experiments, we have used an IoT DDoS malware dataset newly collected by IoTPOT, the first honeypot for collecting IoT threat samples. The malware samples are labelled using VirusTotal with the majority rule. The dataset originally contains 500 malware samples… The number of samples are balanced for each family by randomly removing the samples that belong to classes that are too large. After the preprocessing phase, we analyzed 365 samples where each class has the same number of samples. Among them, we utilize 45 sample (each class has 15 samples) for testing, and the rest for training."
- TRANSFORMATION FUNCTION T(): "We firstly extract one-channel gray-scale images converted from binaries, and then utilize a light- weight convolutional neural network for classifying IoT malware families."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Convolutional neural network
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #68. (Burnap, French, Turner, & Jones, 2018): In 2018, this paper introduced a malware detector that used dynamic analysis features for a self-organizing map (SOM) machine learning model algorithm. They reported an accuracy of 90%.

- CATEGORY: Malware Detector – Dynamic Analysis
- EFFICACY: accuracy of 90%
- TRAINING DATA: "We collected an initial sample of 594 malicious files of the Portable Executable (PE) 32-bit format where at least 10 individual virus scanners labelled the file as malware. For the purposes of developing a 'clean' sample we collected a further 594 files that were labelled 'trusted' - i.e. considered not to be malware by all the individual scanners."
- TRANSFORMATION FUNCTION T(): "During training, a 9-dimensional input vector of normalised values from the training set was directed to either the Good map, used to represent the class of clean baseline data, or to the Bad map, used to represent the class of data that includes 'dangerous' values. Importantly, 'Bad' data will likely include 'Good' examples."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Self organizing feature maps
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #69. (HaddadPajouh, Dehghantanha, Khayami, & Choo, 2018): In 2018, this paper introduced a malware detector that used static analysis and opcode features for a recurrent neural network machine learning model. They reported an accuracy of 98.18%.

- CATEGORY: Malware Detector – Static Analysis and Opcodes
- EFFICACY: accuracy of 98.18%
- TRAINING DATA: "To train our models, we use an IoT application dataset comprising 281 malware and 270 benign ware. Then, we evaluate the trained model using 100 new IoT malware samples (i.e. not previously exposed to the model) with three different Long Short Term Memory (LSTM) configurations"

- TRANSFORMATION FUNCTION T(): Samples are disassembled and opcodes are used as features.
- PRE-PROCESSING FUNCTION P(): Opcode sequence pruning. Statistics are produced from the pruned opcode sequences.
- MACHINE LEARNING MODEL FUNCTION MLM(): Recurrent neural network
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #70. (Azmoodeh, Dehghantanha, & Choo, 2018): In 2018, this paper introduced a malware detector that used static analysis and opcode features for a neural network. They reported a 99.68% accuracy.

- CATEGORY: Malware Detector – Static Analysis and Opcodes
- EFFICACY: accuracy of 99.68%
- TRAINING DATA: "We created a dataset of 1078 benign and 128 malware samples for ARM-based IoT applications [44]. All malware samples were collected using VirusTotal Threat Intelligence platform between February 2015 and January 2017. All goodware were collected from a variety of official IoT App stores such as Pi Store."
- TRANSFORMATION FUNCTION T(): "In this paper, we present a deep learning based method to detect Internet Of Battlefield Things (IoBT) malware via the device's Operational Code (OpCode) sequence… In this study, 4,543 1-gram and 610,109 2-gram distinct OpCode sequences were extracted and CIG(f,CB) and CIG(f,CM) were calculated."
- PRE-PROCESSING FUNCTION P(): Graph generation
- MACHINE LEARNING MODEL FUNCTION MLM(): Neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Method #71. (Pektaş & Acarman, 2018): In 2018, this paper introduced a malware classifier that used dynamic analysis features from a malware sample for a confidence weighted linear classification. They reported an accuracy of 98%.

- CATEGORY: Malware Classifier – Dynamic Analysis
- EFFICACY: accuracy of 98%
- TRAINING DATA: "The testing malware dataset was obtained from 'VirusShare Malware Sharing Platform' [34]. A large amount of malware with different types including PE, HTML, Flash, Java, PDF, APK was downloaded… At the end of the analysis, we prepared a set of testing dataset constituted by 17,400 responding samples belonging to 60 distinct families. Also, 532 benign executables are obtained from the system32 folder of Microsoft windows. These classes are not balanced in terms of the same sample size but since 60 families of malicious executables and one benign class constitute the dataset, balance problem among class size is reduced."
- TRANSFORMATION FUNCTION T(): "Runtime behaviours are extracted with a particular focus on the determination of a malicious sequence of application programming interface (API) calls in addition to the file, network and registry activities. Mining and searching n-gram over API call sequences is introduced to discover episodes representing behaviour-based features of a malware… In this study, we deploy VirMon to extract Windows kernel-level notification routines as the underlying automated dynamic platform and we use Cuckoo to extract API call sequences."
- PRE-PROCESSING FUNCTION P(): Not available
- MACHINE LEARNING MODEL FUNCTION MLM(): Confidence weighted linear classification
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Not available
- POST PROCESSING FUNCTION PP(): Not available

Other Prior Literature

Not all of the prior literature presented classification or detection methods that used machine learning models. A handful of publications provided foundational concepts and experimentation with entropy and they were relevant to this dissertation. This section will briefly identify the additional publications that were reviewed.

In 2007, Lyda and Hamrock examined a structural entropy approach to malware analysis (Lyda & Hamrock, 2007). Note that structural entropy is similar to running window entropy, but there is a skip value used with structural entropy so not every entropy calculation occurs at every index. The authors concluded that decisions could be made with respect to the entropic structure of a malware sample.

Also in 2007, a patent was published that contained the similar material to the Lyda publication above (McMillan & Garman, 2007). This patent described statistical decision making on entropy information of malware. This patent describes how to calculate structural entropy. Running window entropy is a special case of structural entropy where the skip value is one byte. The content of this patent is conceptually similar to the Lyda and Hamrock publication (Lyda & Hamrock, 2007).

In 2011, Sorokin examined the concept of comparing files using structural entropy (Sorokin, 2011). This research used a discrete wavelet transform (DWT) on the structural entropy in order to compare segments. This research demonstrated that structural entropy, of which running window entropy is a special case, can be used to compare the content of files.

In 2013, Baysa examined structural entropy as it relates to metamorphic malware (Baysa, 2013). In this work, Baysa demonstrated that structural entropy can be used to detect changing malware. Later in 2017, Paul examined structural entropy to identify file types (Paul, 2017). Both of these publications provide weight that structural entropy, of which running window entropy is a special case, can be used for malware analysis and possibly classification.

Analysis Of Prior Literature And Contributions To Science

The sheer number of publications relevant to malware detection or classification using machine learning algorithms was unmanageable without a method to organize them. As demonstrated in this chapter, each of the prior methods can be categorized and described using the components of Figure 2 and Definition 1. This generalized model was applied to seventy-one prior methods. This section will explore the content of the prior methods with respect to Malgazer's contribution to science.

In the prior literature, there were forty-one malware detectors and thirty malware classifiers. The following is a list of all the malware detectors:

- Method #01 – Method #12
- Method #16
- Method #17
- Method #20
- Method #21
- Method #22
- Method #23
- Method #24
- Method #26
- Method #27
- Method #30
- Method #32
- Method #37
- Method #39
- Method #42
- Method #43
- Method #46
- Method #47
- Method #48
- Method #49
- Method #52
- Method #53
- Method #55
- Method #62
- Method #63
- Method #64
- Method #65
- Method #68
- Method #69
- Method #70

Malgazer is different than all of the previously listed methods because they are malware detectors and not general classifiers. As shown in Figure 1, all detectors are a subset of classifiers. As by the research question posed in the introduction chapter, Malgazer will be classifying samples after the detection has already occurred. Therefore, malware classifiers are more relevant to Malgazer. The following list contains the thirty malware classifiers surveyed in this chapter:

- Method #13
- Method #14
- Method #15
- Method #18
- Method #19
- Method #25
- Method #28
- Method #29
- Method #31
- Method #33
- Method #34
- Method #35
- Method #36
- Method #38
- Method #40
- Method #41
- Method #44
- Method #45
- Method #50
- Method #51
- Method #54
- Method #56
- Method #57
- Method #58
- Method #59
- Method #60
- Method #61
- Method #66
- Method #67
- Method #71

These thirty methods attempt to accomplish the same goal as Malgazer, which is to classify malware samples. From the thirty classifiers above, only two classifiers use features similar to running window entropy: Method #13 and Method #20. Even though it had a high accuracy, Method #13 only classified samples if they were packed. This was not the only intention of Malgazer because the classifications will be made with respect to functionality. On the other hand, Method #20 only reported an accuracy of 87.2%. This accuracy could likely be improved based upon accuracies reported in other methods, as the next paragraph will explain. Method #20 also used structural entropy, of which running window entropy is a special case. This dissertation examines running window entropy as a feature space. Therefore, it is obvious that the research question Malgazer is trying to solve contributes to science on a topic that has only two similar prior publications. If Malgazer is able to functionally classify malware using running window entropy and machine learning, it will be a contribution to science where prior literature is lacking.

It should be noted that there were five detectors that included entropy as features: Method #22, Method #23, Method #26, Method #27, and Method #55. The efficacy of these detectors varied widely from 68.7% to 99% accuracy. It is clear from these methods that if detectors can reach up to 99% accuracy with entropy, classifications could be improved from 87.2% accuracy seen in Method #20. The Malgazer classifier improved accuracy to 95% using running window entropy.

Selection Of Prior Works For Empirical Comparison

Efficacy, in general, varied greatly across the seventy-one methods. Accuracy was as low as 60% up to a reported perfect accuracy. Accuracy can be biased by any number of factors in the machine learning model training and testing phases, therefore accuracy dependencies need to be removed with an empirical study. This dissertation will compare an empirical study of a prior method with an empirical study of Malgazer. This study will attempt to remove dependencies that may be biasing the reported accuracies of prior methods. Unfortunately, most of the methods in prior literature did not include the training data sets, the testing data sets, source code, or enough details about the method so that the experiments could have been replicated with some degree of certainty. The “GIST” method, Method #14, was chosen from

many when an author of the paper replied to a request for more information that allowed for the replication of their method.

Since no method has components from Figure 2 and Definition 1 that are the same as Malgazer, it was appropriate to select a similar prior method for empirical comparisons. Method #14, also referred to more compactly as "GIST" throughout this paper, was implemented using the same datasets and methods used to develop Malgazer. This process allowed for an in-depth empirical comparison between Malgazer and "GIST". This author developed Python source code of the "GIST" method based upon an e-mail conversation with one of the authors and the literature cited in Method #14.

Summary

This chapter methodically reviewed the prior literature on machine learning based malware classification and detection. The components from Figure 2 and Definition 1 were used to describe each prior method. Each prior method was assigned a method number and a citation for easier referencing throughout this dissertation. An analysis was conducted on the prior literature and it was discovered that only two publications addressed the same topic as Malgazer, thereby showing Malgazer's contribution to science. The two publications did not use the same methodology as Malgazer, so Malgazer is new science. This chapter also reviewed prior relevant works that were foundational. Lastly, this chapter discussed the selection of a prior Method #14, also referenced through this dissertation as "GIST", for the in-depth empirical comparison to Malgazer in future chapters.

CHAPTER 3

RESEARCH METHODOLOGY

Introduction

This chapter will present the research methodology for the development of Malgazer and answer the research question from the first chapter. Building Malgazer was a complex endeavor and the research methodology will be presented chronologically by milestone to highlight each design characteristic. The research methodology for Malgazer consisted of twelve distinct phases presented in Figure 9, and each phase will be described in this methodology chapter as a section. Then, this chapter will conclude by discussing how this research methodology accomplished the research objectives presented in the first chapter.

First, Malgazer needed running window entropy data but the original algorithm was too slow to compute tens of thousands of samples during experimentation. Therefore, the running window entropy calculation was optimized for time. This optimization is the content of chapter 4. Simultaneously, an exploratory data set was sourced from VirusTotal using a methodology described later in this chapter.

The output of the running window entropy algorithm produced a data set that could be used as input features to machine learning models, but the known classifications for each sample were needed next. Publicly available classifications were collected for this purpose from VirusTotal. Next, the VirusTotal data and running window entropy data was explored qualitatively to determine the best approach to answer the research question. The qualitative exploration will be mentioned here, but the results are described in more detail within chapter 5. Based upon those findings, the final malware data set was sourced. The final malware data set were used to train and test the classifiers developed throughout this research.

During the exploration phase, it was discovered that classification data from VirusTotal was not consistent between vendors, so an algorithm was developed to produce unified

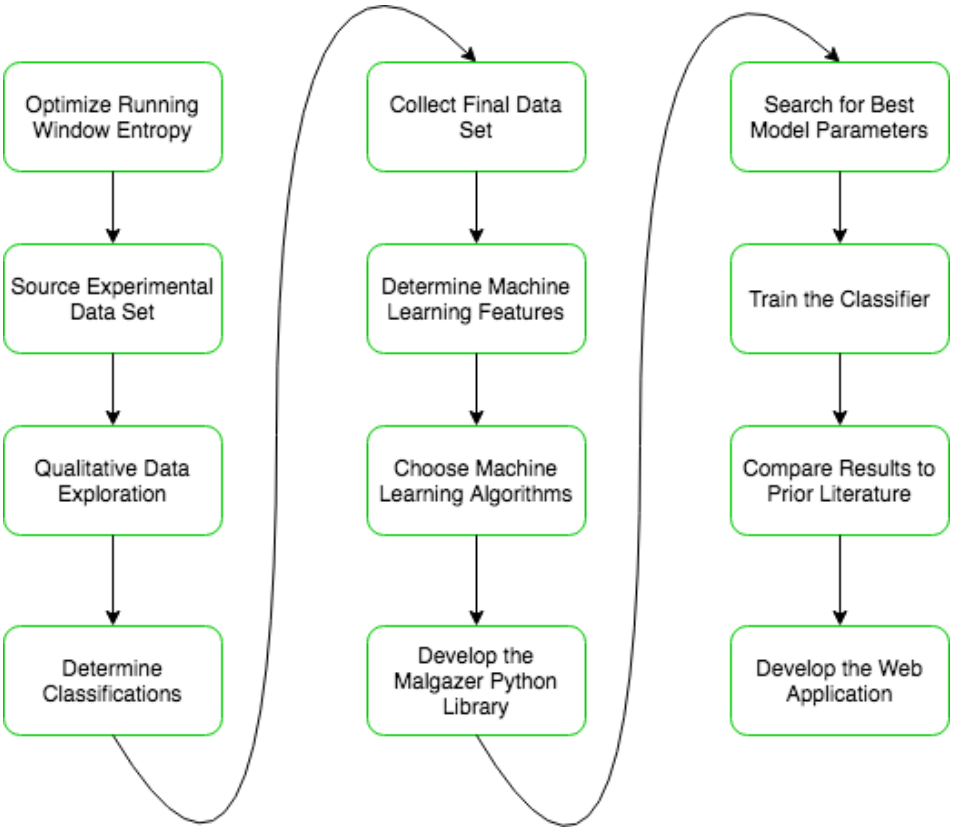


Figure 9. The design process

functional classifications from the VirusTotal data. The algorithm to determine classifications from VirusTotal data will be presented as a section later in this chapter.

Next, the feature data (running window entropy) was computed for the final training data set using the Malgazer Python library. This chapter will include a short explanation of this library and a tour of the source code repository is available in appendix A. The machine learning algorithms selected for this research will also be explained in this chapter.

Next, the optimal hyper-parameters were identified using a methodical grid searching technique, as discussed in the introduction chapter. The searching technique and the results are discussed further in chapter 6. Then, the optimal hyper-parameters calculated from grid searching were used to cross fold validate (train) the Malgazer classifier to measure mean accuracies and accuracy variance. Cross fold validation will be explained in this chapter and the results of the validation are presented in chapter 6. This chapter will also present the methods used to compare the Malgazer classifier with prior literature empirically. The data, analysis of the data, and comparison to prior methods are presented in chapters 6 and 7. After the Malgazer classifier was developed, a web application was written so the classifiers could be easily

accessed remotely with very little programming familiarity. The web application is explained in chapter 8. The next sections within this chapter will address each item summarized in this introduction.

Optimizing Running Window Entropy

When initially exploring this research topic and experimenting with malware data samples in early 2017, it was quickly obvious that computing running window entropy at scale for tens of thousands of samples, or more, was going to be a slow endeavor. Therefore, one of the first goals and first completed milestone was to optimize the original running window entropy algorithm with respect to time so that it would calculate faster. It was obvious that most of the computation time included the log function from Shannon's equation, and that function was called repeatedly within nested "for" loops. Chapter 4 describes the development of a faster algorithm that flattens the nested "for" loops to decrease the total number of log function calls. This milestone led to the faster computation of running window entropy so that the data sets discussed in the next sections could be calculated by a single individual with limited computing resources. The measurement of success for this design science research artifact is also addressed in Chapter 4.

Determining Classifications

Now that the running window entropy data could be calculated efficiently, classifications of the malware samples were required. There was no one source that provided a unified and reliable set of functional classifications, so a method was needed to obtain them. This section will describe how functional classifications were determined from VirusTotal ("VirusTotal," n.d.-a) public information.

VirusTotal is very well known in the malware analysis community and it provides a free scanner for malware samples using numerous anti-virus and anti-malware products. VirusTotal consumes multiple file samples every second from all over the world, of which many are malware. VirusTotal aggregates a continual, up to date malware sample feed from their users' uploads that can be used to source malware for machine learning algorithms.

VirusTotal provides the public access, commercially, through an “Intelligence” feed of this sample stream. This author had commercial access to the intelligence feed during this research as part of his regular employment duties, although the methods described within this dissertation do not add on costs to the intelligence feed’s base access like other VirusTotal API methods. Through VirusTotal, the malware ingestion feed can be searched live by user generated Yara (VirusTotal, n.d.-b) signatures. Yara is an arbitrary data matching application that is used heavily within the malware analysis community. Any VirusTotal samples that matched a Yara signature are stored for the user to retrieve as a notification at a later time. It is through this facility that this author created the functional classifications along with all of the data sets.

When a malware sample matches a provided Yara signature, the results of several anti-virus and anti-malware product scans of the sample are also provided in the report returned to the user. In this research, the results of the VirusTotal malware scanning was used to determine the functional classification of each malware sample. Through experimentation, it was found that VirusTotal would return approximately seventy anti-virus or anti-malware product scans for each submitted malware sample in early 2018. Some detections in the exploratory data set were better than others for automated classification purposes, and this will be discussed further in chapter 5.

Through visual inspection, it was obvious that each product in VirusTotal used independent classifications schemes. For example, one product in the report may have classified a sample as “Ransomware” while another product classified the same sample as “Locky”. Making matters worse, some products in the report used a functional classification and family name, but not both at the same time for a sample. Since this research was focused on functional classifications, this made sourcing consistent functional classifications difficult. The products returning different classifications for the same sample may each have been correct in their assessment, but a unified functional classification must be drawn from the numerous products available on VirusTotal if that data is to be used for machine learning purposes. This is an example of the transformation “T” and pre-processing “P” functions in the generalized machine learning based malware classification model introduced in the first chapter where the data it transformed/preprocessed was the VirusTotal file classification. The VirusTotal classification data had to be normalized and standardized before it was useful to machine learning algorithms.

In order to determine a unified classification system based upon the VirusTotal reports, an experimentation malware data set was sourced. The exploratory data set was captured using the following Yara signature[3] on the VirusTotal Intelligence feed in mid-March 2018:

```
01.import "math"
02.import "pe"
03.
04./* 20180312.1 */
05.
06.rule peexe_files_20180312_1 {
07.  condition:
08.    file_type contains "peexe"
09.}
```

The rule above filtered the intelligence feed and only captured samples that were Window PE executable type. For this research, the results from the rule were only retrieved for samples that had at least twenty-five detections in VirusTotal. The results were gathered using scripts freely available in the Malgazer source code repository (Jones, 2017b). Returned from this filter was a data set of VirusTotal classifications of approximately 28,042 samples. Each sample included scans from the seventy products, listed below:

- ALYac
- AVG
- AVware
- Ad-Aware
- AegisLab
- AhnLab-V3
- Alibaba
- Antiy-AVL
- Arcabit
- Avast
- Avast-Mobile
- Avira
- Babable
- Baidu
- BitDefender
- Bkav
- CAT-QuickHeal
- CMC
- ClamAV
- Comodo
- CrowdStrike
- Cybereason
- Cylance
- Cyren
- DrWeb
- ESET-NOD32
- Emsisoft
- Endgame
- F-Prot
- F-Secure
- Fortinet
- GData

[3] Note that line numbers were added for convenience and are not part of the signature.

- Ikarus
- Invincea
- Jiangmin
- K7AntiVirus
- K7GW
- Kaspersky
- Kingsoft
- MAX
- Malwarebytes
- McAfee
- McAfee-GW-Edition
- MicroWorld-eScan
- Microsoft
- NANO-Antivirus
- Paloalto
- Panda
- Qihoo-360
- Rising
- SUPERAntiSpyware
- SentinelOne
- Sophos
- Symantec
- SymantecMobileInsight
- Tencent
- TheHacker
- TotalDefense
- TrendMicro
- TrendMicro-HouseCall
- VBA32
- VIPRE
- ViRobot
- Webroot
- Yandex
- Zillya
- ZoneAlarm
- Zoner
- eGambit
- nProtect

Upon visual inspection, the following six products generally provided the most consistent functional classification name somewhere within their classification:

1. Microsoft
2. Symantec
3. Kaspersky
4. Sophos
5. Trend Micro
6. ClamAV

The final functional classification was determined by the classification of the products in the preceding list, in increasing order. The lower number in the previous list took precedence over products with higher numbers, such that Malgazer used the Microsoft product classification before the ClamAV product classification for each sample.

Once the functional information had been parsed from the original VirusTotal product classification string, the data was normalized to just six malware groups to focus the scope of this research. The classifications chosen for this research were Trojan, Ransom, Worm, Backdoor, Potentially Unwanted Application (PUA), and Virus. The choice of these groups

Table 1. Malgazer classification lookup table

Search String	**Malgazer Classification**
Trj	Trojan
PUP	PUA
Wrm	Worm
Bkdr	Backdoor
Cryp	Ransom
PUA	PUA
RiskTool	PUA
VirTool	Virus
VirLock	Ransom
HackTool	PUA
RemoteAdmin	PUA
Trojan	Trojan
Worm	Worm
Troj	Trojan

came from an exploratory data analysis phase with the initially collected exploratory data set, mentioned previously. This exploratory data analysis will be discussed in a later chapter.

After the classifications were determined by the six products in the list above, they were case-insensitively searched for strings using the terms from the first column of Table 1. The translation in Table 1 came from a best effort attempt to understand and classify the exploratory data set captured from VirusTotal. The existence of a string in the first column applied a classification from the six malware groups listed in the second column of Table 1, where the six products were searched in order (Microsoft, Symantec, etc.). This functional classification was used as Malgazer's classification for the next phases of this research.

Qualitative Data Exploration

After downloading and classifying approximately 28,042 samples from VirusTotal in March 2018, a brief data exploration effort began. During this exploration phase, the running window entropy data was computed over every sample for four windows: 512; 1,024; 2,048; and 4,096 bytes. The running entropy data was translated into data structures commonly used for machine learning through the development of the Malgazer Python library.

After some statistical calculations, it quickly became apparent that the exploratory data was unbalanced. Unbalanced data meant there were more samples of one category versus others so that they were not distributed evenly in a histogram. This is a common situation because the numbers of threats per classification is dynamic as the threat landscape changes over time. During one era, ransom malware may have been more popular than another era, for example. Therefore, it was determined that this exploratory data set will only be used to verify that the running window entropy can be used to classify samples using machine learning methods and aid in writing the Malgazer Python libraries (explained in an upcoming section). This path was chosen because it was likely that the accuracies discovered during the exploratory phase could have been artificially high due to learning bias. Instead, it is important to know that an exploratory data analysis phase was used to make the initial design decisions discussed in this chapter. More information about the data exploration process is presented in chapter 5. Since the exploratory data set was unbalanced, a larger final and balanced data set was needed and the sourcing methodology for the data set is presented in the next section.

Collecting The Final Data Sets

After reviewing the classifications in the exploratory phase, the larger final data set was sourced from VirusTotal using several Yara signatures throughout May of 2018. The classifications were determined by the classification determination algorithm previously described in this chapter. Each of the Yara signatures was allowed to run until approximately 11,000 samples were collected from each of the six malware classification groups: Backdoor, Worm, Trojan, Virus, PUA, and Ransom. A sample was kept if twenty-five or more products on VirusTotal determined the sample was malware, otherwise the sample was discarded. Then, each of the binaries was downloaded from the data set until there were 10,000 samples left in each classification, making the complete data set 60,000 malware samples. The Yara signatures used for this final data set collection follows.

Backdoor Samples

```
01.import "math"
02.import "pe"
```

```
03.
04./* 20180501.1 */
05.
06.rule peexe_backdoor_files_20180501_1 {
07.  condition:
08.    (file_type contains "peexe") and (microsoft contains
   "Backdoor")
09.}
```

Worm Samples

```
01.import "math"
02.import "pe"
03.
04./* 20180501.1 */
05.
06.rule peexe_worm_files_20180501_1 {
07.  condition:
08.    (file_type contains "peexe") and (microsoft contains
   "Worm")
09.}
```

Trojan Samples

```
01.import "math"
02.import "pe"
03.
04./* 20180501.1 */
05.
06.rule peexe_trojan_files_20180501_1 {
07.  condition:
08.    (file_type contains "peexe") and (microsoft contains
   "Trojan")
09.}
```

Virus Samples

```
01.Virus Files
02.
03.import "math"
04.import "pe"
05.
06./* 20180501.1 */
07.
08.rule peexe_ransom_files_20180501_1 {
09.  condition:
```

```
10.    (file_type contains "peexe") and (microsoft contains
   "Ransom")
11.}
```

PUA Samples

```
01.PUA Files
02.
03.import "math"
04.import "pe"
05.
06./* 20180508.1 */
07.
08.rule peexe_pua_files_20180508_1 {
09.  condition:
10.    (file_type contains "peexe") and (signatures contains
   "PUA" or signatures contains "PUP")
11.}
```

```
01.import "math"
02.import "pe"
03.
04./* 20180501.1 */
05.
06.rule peexe_hacktool_files_20180501_1 {
07.  condition:
08.    (file_type contains "peexe") and (microsoft contains
   "HackTool")
09.}
```

Ransom Samples

```
01.import "math"
02.import "pe"
03.
04./* 20180508.1 */
05.
06.rule peexe_ransom_files_20180508_1 {
07.  condition:
08.    (file_type contains "peexe") and (signatures contains
   "Ransom")
09.}
```

Machine Learning Algorithm Input Features

Two types of data were computed and used as machine learning features in this study: running window entropy and GIST. The GIST features were described in Method #14 of the literature review chapter and were used for an empirical comparison to Malgazer's running window entropy features. As discussed previously, the running window entropy data is variable length depending on the malware's size and the running window size. Therefore, the arrays had to be resampled to a fixed number of values for every sample in the data set. Four values were used for the running window entropy data sampled length. The running window size for the entropy calculations can also vary, so four values were chosen as the window size. The final data set was computed with the running window entropy window sizes of 256; 512; 1,024; and 2,048 bytes sampled to 512; 1,024; 2,048; and 4,096 values. The final data set was also computed with the standard 320 features as described in the GIST Method #14. All of the features were computed using the Malgazer Python libraries.

The Malgazer Python Libraries

Several Python libraries were created to compute the features in the data sets and are referred to as the Malgazer Python libraries. The Malgazer Python libraries are located in the Malgazer source code respository (Jones, 2017b). There are four Python modules within this repository. Each will be explained below. A tour of the Malgazer repositories is available in appendix A.

entropy.py

This module includes a "RunningEntropy" class that was used to calculate running window entropy from data. This module also provided a resample function so that running window entropy could be resampled to a fixed number of data points.

files.py

This module includes a "Sample" class that was used to hold the data of a malware sample. The malware sample data could be read from a file or a list of bytes in memory, and

the “Sample” class also calculated the GIST data, running window entropy, and SHA256 hash values.

ml.py

This module includes a “ML” class that is a wrapper around machine learning algorithms. This class was used to create, train, use, predict, and measure various machine learning algorithms.

utils.py

This module includes a “Utils” class that provided various useful functionalities to Malgazer. For example, all of the preprocessing functionality that estimated classifications based upon raw VirusTotal reports is located here. Other external Malgazer scripts actually call various functions within this “Utils” class for most of the logic.

Choosing Machine Learning Algorithms

Python is used heavily by the data science community, so there were several machine learning algorithms already implemented. All of the machine learning algorithms described in the introduction chapter were chosen for this study and they were already implemented either in SciKit-Learn (“scikit-learn,” n.d.) or Keras (“Keras,” n.d.) with Tensorflow (“Tensorflow,” n.d.) as a backend. These algorithms were chosen because it was a manageable number of machine learning algorithms that were also cited frequently in prior literature.

Searching For Best Model Parameters

One of the machine learning algorithms written into Malgazer is the grid searching algorithm. The grid searching algorithm reads possible hyper-parameters of the machine learning model from the user and trains the machine learning model with all of the possibilities, reporting them in descending order of accuracy. This allows for orderly searching of model parameters to more finely tune the machine learning model to the input data. For example, as discussed in the introduction chapter, neural networks can be built with

different numbers of units in different numbers of hidden layers with different optimizers and activation functions. The structure of the neural network may play a great role for one data type, but not another data type. The chosen optimizer may converge for one training, but not another training. Using the grid searching algorithm, one can iterate through all of the possible neural networks and select the one with the highest accuracy.

For other models, such as SVM, there are multiple hyper-parameters that should be tuned according to the data. Grid searching locates the best accuracy across an n-dimensional space, where each dimension is defined by the variable representing a hyper-parameter (such as the variable "C"[4] in an SVM model, or the number of layers in a neural network). Once the hyper-parameters were optimized for the final data set, the subsequent trainings of the Malgazer and GIST classifiers used those optimal parameters.

Training Machine Learning Models

Once the optimal hyper-parameters were calculated, they were used as the hyper-parameters to train the final machine learning models. Two classifiers were created for each type of machine learning model, a Malgazer classifier and the GIST classifier. These classifiers provided a comparison of empirical results to one another, and a theoretical comparison other reported accuracy in prior literature. The specific information used to train the classifiers is presented in chapter 6, along with the training results.

In general, the studied classifiers were trained using two methods. First, each classifier was trained using cross fold validation. In this research, ten folds were used for the cross-fold validation. This means that the final data set, discussed previously, was assembled into ten non-overlapping data sets where training occurred with nine of the folds and tested on the tenth fold. The randomly selected samples were stratified, meaning that each class was represented equally in each fold, removing as much bias as possible. Then, the process was repeated ten times until every fold has been used as the testing set once. For each fold, the accuracies were recorded so that the mean and variance could be computed. Cross fold validation is common in data science literature and helps provide a more realistic accuracy score for a classifier.

[4] This is a different "C" than in the generalized malware classification model presented in the first chapter. The "C" discussed here is a scalar floating point value.

Once the cross-fold validation scores were recorded, the classifiers were trained one final time on 90% of randomly selected samples from the final data set and tested on the remaining 10%. Again, the randomly selected samples were stratified, meaning that each class was represented equally in the training and testing data sets. Training the classifier one final time also saved the classifier as a file that can be used in the web application developed during this research and described later. The final training also provided an ROC curve with AUC data.

Comparison Of Malgazer To Prior Literature

The classifiers created in the training phase were measured using AUC (Area under the curve of ROC) and confusion matrices. Confusion matrices allowed for the computation of many other statistics, such as accuracy. Overall accuracy was used for the major comparison to prior literature, but AUC was also be examined where appropriate. Empirical results were obtained during the classifier training phase and are presented in detail in chapter 6. Although this author did not have access to every data set for the methods described in the literature review chapter for a fair comparison, a theoretical comparison to their reported accuracies is made in chapter 7 nonetheless. Analysis regarding accuracy per class for the classifiers will also be explored in chapter 7.

If Malgazer performed better than random guessing, which has a 1/6 probability of being right in the six classification groups chosen here, then it answered the design science-based research question in the first chapter. If Malgazer did not perform better than random guessing, then the null hypothesis was true. If an instance of a Malgazer classifier performed better than random guessing, the existence of such classifier would not only answer the research question, but it would demonstrate that the null hypothesis is false. While it is difficult to measure the true efficacy of all design science artifacts, such as the generalized model in Figure 2 and Definition 1, instantiations in the form of a classifier can be measured empirically. Later chapters discuss the measured comparisons to prior methods.

The Web Application

Once the Malgazer and GIST classifiers were instantiated, they still required someone with programming knowledge to use them. Therefore, a web application that provides access for those without programming knowledge was developed. For those with programming knowledge, there was an API exposed in the Malgazer web application that allows for programmatic access to the classifier. This makes use of the classifier much easier than as raw data structures in Python. The web application was the last design science artifact created by this research, and it can be found in the Malgazer source code repository.

Accomplishing the Research Objectives and Contributions

Several research objectives were presented in the first chapter, and they will be revisited here to discuss how this research methodology accomplished those goals. First, a generalized machine learning based malware classifier model was presented. This model was used to describe the prior literature in the previous chapter. As a result, the model was validated by demonstrating its usefulness categorizing prior literature, plus it was be used when developing the Malgazer classifier in chapter 6.

Next, an optimized running window entropy algorithm was discussed. The optimization process is the content of the next chapter. The next chapter will show that the running window entropy was optimized so that the calculations were faster. Next, a method to collect and normalize functional classification information from VirusTotal was discussed. This chapter presented such a method and accomplished this objective.

After that, a method to scale the training of machine learning based classifier using cloud resources was discussed. As part of chapter 6, this method will be presented. Then, the first chapter discussed an optimized machine learning based malware classifier. Developing this classifier is the basis of chapter 6, and with it the data sets, logs, and figures have been included online for independent verification and replication. Lastly, the introduction discussed a user web application to automate the machine learning based malware classifier. This application will be discussed in chapter 8. Therefore, every objective defined in the introduction has been covered with this research methodology chapter or will be covered in future chapters.

Summary

This chapter discussed the research methodology used to create the design science research artifacts introduced in the first chapter. This chapter provided the design decisions and the logic behind them, such as optimization of running window entropy or the method to normalize and standardize VirusTotal data to create the six Malgazer functional classification groups. After a qualitative exploration of malware sample data, a final data set was sourced from VirusTotal using Yara signatures presented in this chapter.

Then, the types of features from running window entropy and GIST were discussed. These features were calculated using the Malgazer Python library, and each module from this library was also discussed in this chapter. Next, the choice of machine learning algorithms was made for this research project, and the background for these algorithms was initially discussed in the introduction chapter. Once the algorithms were chosen, grid searching was used to find the optimal hyper-parameters for each algorithm for the final data set.

The final Malgazer and GIST classifiers were trained on the final data set, and the results of the training were compared to prior literature. The results of the training and the comparison are presented in chapters 6 and 7. Finally, a web application was developed to provide easy and programmatic access to the trained classifiers. This was the last design science artifact created by this research.

This chapter concluded by revisiting the research objectives and discussing how each one was satisfied by this research methodology. The next chapter will discuss the results of the qualitative data exploration process.

CHAPTER 4

OPTIMIZING RUNNING WINDOW ENTROPY

Introduction and Purpose

This dissertation examines running window entropy as a feature set to a machine learning based classifier. Machine learning classifiers are trained and tested on large training data sets. To reduce costs associated with this computationally intensive operation, the running window entropy algorithm was optimized for this research and is presented in this chapter. The material in this chapter was first published at a peer reviewed conference by Jones and Wang, available at (Jones & Wang, 2018). Slight modifications were made from the original paper to read more formally and fit into this dissertation, but the design science artifact remains the same.

Evaluation Criteria

The evaluation criteria for successful optimization will be a comparison of running time for the original algorithm versus the optimized algorithm theoretically and empirically. For empirical measurements, the algorithms presented in this chapter were implemented in Python, which is a platform independent language. This test could be performed with other languages and with similar constructs should produce similar results, with a factor of speed associated with that particular programming language. This chapter will present a Python implementation, but the focus of this chapter is on the algorithm itself. The algorithm, if implemented with similar constructs discussed in this chapter and the associated Python code, will be shown to be optimized theoretically regardless of the implementation language.

The algorithms will be measured solely on their computing time, removing any factors associated with multi-processer or multi-threaded systems. This measurement is available in Python through the “time.process_time” function. The running time will be captured for the execution of each algorithm ten times in a row and then the average will be computed. A noticeable decrease in the average running time of one algorithm over another will be

considered proof of time optimization for the second algorithm. To keep the comparisons simple, the algorithms will be implemented in a single threaded, single processor fashion. The execution times for both algorithms in this chapter were measured on an Apple Macbook Pro with 16 GB 1.6 GHz DDR3 RAM, a 2.8 GHz Intel Core i7 processor, running on MacOS Sierra v10.12. The correctness of the algorithms was verified using code this author developed, openly available at (Jones, 2017a).

The Original Algorithm

Shannon defined (Shannon, 1948) entropy for an information source as:

$$H = -K \sum_{i=1}^{n} p_i log p_i \quad (2)$$

The malware sample's size is fixed with length *k* and will be defined as:

$$A = \{a_1, a_2, \ldots a_{k-1}, a_k\} \quad (3)$$

A data window of length *j* that begins at offset *l* within *A* will contain a portion of data for *A*, such that:

$$X_l = \{a_l, a_{l+1}, \ldots a_{l+j-1}\} \quad (4)$$

The running window entropy can be visualized in Figure 10 with a four-byte running window size, for demonstration purposes only. The figure demonstrates the nested "for" loops in the calculation and the output of one window entropy value H_2.

Entropy values will be calculated for each X_l where $1 \leq l \leq k - j + 1$. This is the window that will be defined as the running window entropy. The result of the running window entropy will output the following series:

$$W = \{H_1, H_2, \ldots H_{l-1}, H_l\} \quad (5)$$

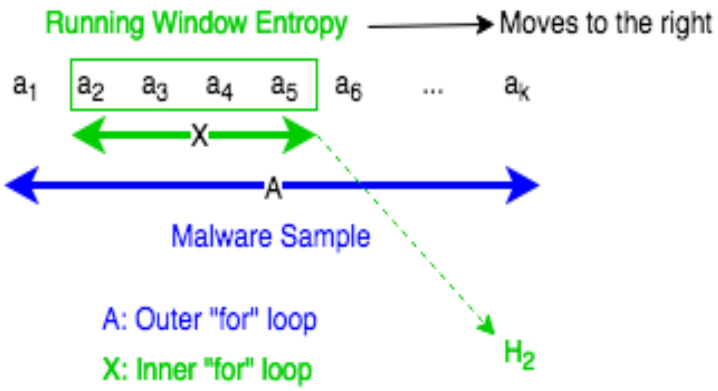


Figure 10. Running window entropy revisited

This equation can be restated in terms of j, the size of the running window, instead of l as follows:

$$W = \{H_1, H_2, \ldots H_{k-j}, H_{k-j+1}\} \quad (6)$$

Note that Shannon's entropy equation assumes there is a continuous probability function for the information. Since data is discrete and not continuous, a frequency sample will be used to estimate the probabilities (Lall, Sekar, Ogihara, Xu, & Zhang, 2006).

$$H = -\sum_{i=1}^{n} \frac{m_i}{j} log_2(\frac{m_i}{j}) \quad (7)$$

Here, m_i is the total count of the symbol i within the running entropy window. j is the size of the running entropy window. There are n possible values for the symbol. When the symbol is defined to be a byte, as it is in this chapter, there are 256 total symbols. Therefore, the equation above can be simplified to:

$$H = -\sum_{i=1}^{256} \frac{m_i}{j} log_2(\frac{m_i}{j}) \quad (8)$$

When m_i is zero, note that the *i*-th component is zero so that the *log* of zero, which is undefined, can be ignored completely.

The result from equation 8 will be between 0 and 8. A value of 0 means the data is not random, while a value of 8 means that the data is maximally random. Assuming the entropy window is at least 256 bytes, dividing the result of the equation by 8 will normalize the entropy values between 0 and 1. This is an example of K from Shannon's equation equaling $\frac{1}{8}$ for the purposes of normalization. For entropy windows less than 256 bytes, this scale constant could be the ceiling of log base 2 of the window size to normalize it between 0 and 1, but windows less than 256 bytes will not be considered in this dissertation.

The running window entropy calculation described in equation 8 can be implemented with the following Python function, also available at (Jones, 2017a, 2017b):

```
01.# Original Window Entropy Function
02.def original_window_entropy():
03.    # Start the time counter
04.    starttime = time.process_time()
05.
06.    # Entropy list
07.    H = [0 for i in range(1, k-j+2)]
08.    for x in range(1, k - j + 2):
09.        m = [0 for i in range(0, 256)]
10.        for y in range(1, j + 1):
```

```
11.             m[malware[x + y - 2]] += 1
12.         entropy = float(0)
13.         for y in range(0, 256):
14.             if m[y] != 0:
15.                 entropy += -(float(m[y] / j) *
   math.log2(float(m[y] / j)))
16.         H[x - 1] = entropy / K
17.     # End the time counter
18.     endtime = time.process_time()
19.     return endtime-starttime, H
```

In the algorithm above, lines 8, 10 and 13 form a nested "for" loop over the data: an outer "for" loop for the malware file size and inner "for" loops for the running window entropy size. Throughout this chapter, the inner "for" loops can be identified as "the inner for loop" because two inner "for" loops can be represented by one inner "for" loop, in basic time complexity analysis, times a constant multiplier. The nested "for" loops are a time-consuming task that basic theoretical algorithm analysis would determine to be tilde a factor of $k*j$ in which k is the size of the malware (in bytes) and j is the size of the entropy window (in bytes). A tilde notation of an algorithm is the asymptotic growth approximation of the algorithm's running time as a factor of input. In other words, the tilde notation of an algorithm approximates the growth of running time for the algorithm as a function of the size of the input. In this case, the time growth is a function of the window size and the malware size.

Using the evaluation criteria discussed previously, ten executions of the original algorithm were measured and averaged for malware sizes 256 bytes through 131,072 bytes (128KB). The malware in this case was simulated by an array of random byte values. The choice of a simulated malware file does not impact this algorithm's performance and the same arrays were used for testing both algorithms. The simulated malware files were also measured for entropy window sizes 256, 512, 1,024, and 2,046 bytes. Note that it is not possible to calculate the running window entropy on a malware file with a size less than the window. The measured data for the original algorithm is available in Table 2.

Table 2. The original RWE algorithm execution time in seconds

Malware Size (bytes)	256 Byte Entropy Window	512 Byte Entropy Window	1,024 Byte Entropy Window	2,048 Byte Entropy Window
256	1.64E-04	N/A	N/A	N/A
512	4.09E-02	2.51E-04	N/A	N/A
1,024	1.26E-01	1.22E-01	3.56E-04	N/A
2,048	2.92E-01	3.76E-01	3.76E-01	5.69E-04
4,096	6.35E-01	8.91E-01	1.12E+00	1.20E+00
8,192	1.39E+00	1.96E+00	2.68E+00	3.64E+00
16,384	2.75E+00	4.10E+00	6.32E+00	8.71E+00
32,768	5.63E+00	8.38E+00	1.23E+01	1.84E+01
65,536	1.19E+01	1.69E+01	2.48E+01	3.86E+01
131,072	2.37E+01	3.39E+01	5.24E+01	7.86E+01

Figure 11. The original RWE algorithm performance in seconds

The data from Table 2 was plotted two-dimensionally over the malware size and running window size in Figure 11. Time is clearly dependent on the size of the malware sample and running window, increasing for both dimensions. This supports the theoretical tilde analysis of the algorithm showing that the asymptotic growth of the algorithm is a factor of $k*j$, the window length times the malware size.

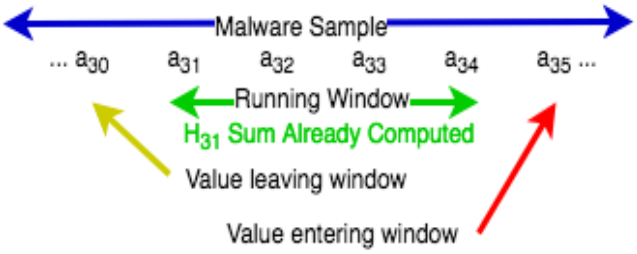


Figure 12. Running window entropy optimization

The Optimized Algorithm

The running window entropy algorithm can be optimized by flattening (in other words, removing) the inner "for" loop, represented by lines 10-15 in the algorithm presented above. The optimization takes advantage of prior calculations within the running window entropy, using the previously computed values from the entropy sum at the previous offset, and only correcting the running sum for differences caused by the new byte entering the window from the right and the old byte leaving the window out of the left when the window moves to the next offset. This process can be visualized graphically in Figure 12 with a four-byte running window size for demonstration purposes only. The old value leaving the window on the left is represented by the gold arrow. The new value entering the window on the right is represented by the red arrow. The computed sum for the current entropy at offset 31 is represented in green. This is the sum that will be reused in the optimization for offset 32. In other words, the optimization corrects the already computed sum in green with the change in value for the removed gold and added red arrows and keeps the other prior computations already completed in the green value.

With only four operations, the entire inner "for" loop can be removed. The four operations replacing the inner loop are: 1) remove the entropy of the symbol leaving the window on the left (gold arrow), 2) decrement the symbol count and add back the entropy (green arrow) of the newly decremented count of the symbol leaving on the left (gold arrow), 3) remove the entropy (green arrow) for the symbol entering on the right (red arrow), 4) update the symbol count and add back the entropy (green arrow) of the newly increased count of the symbol entering on the right (red arrow). To help understanding the optimized algorithm, the following pseudocode explains the optimization:

1. Fill a first in, first out queue with a data window X_l at $l = 1$ where the left side of the queue is the first inserted data value and the right side is the last inserted data value.
2. Count the number of each symbol in X_l and save the results into an array of symbol counters named S.
3. Calculate the frequency of each symbol in S by dividing each count value for the symbol by the entropy window length j. Normalize the data, if desired, by multiplying the results by K.
4. Calculate each $\frac{m_i}{j}$ from S to calculate the entropy as described by the entropy equation number 8. Save this single entropy result in a variable named C.
5. Output C into W as H_1.
6. Save this C as the running sum for the next steps.
7. While $l \leq k - j + 1$ $l \leq k\text{-}j + 1$do:
8. Pop the left most value off $X_l X_l$.
9. Add $K \frac{m_i}{j} log_2(\frac{m_i}{j})$ to C, where m_i is the value popped off the queue X_l in the previous step.
10. Decrement the value counter for this popped data value from S.
11. Subtract $K \frac{m_i}{j} log_2(\frac{m_i}{j})$ from C, where m_i is the decremented value count representing the value popped off the queue X_l in the previous steps and now in *S*.
12. Push the next value from A into the right most value of X_l.
13. Add $K \frac{m_i}{j} log_2(\frac{m_i}{j})$ to C, where m_i is the value pushed into the queue X_l in the previous step.
14. Increment the value counter for this pushed data value from S.
15. Subtract $K \frac{m_i}{j} log_2(\frac{m_i}{j})$ from C, where m_i is the incremented value pushed into the queue X_l in the previous step and now in S.
16. Output C into W as H_l.
17. Save this C as the running sum for additional steps.
18. Let $l = l + 1$.
19. Go to step 7 if $l \leq k - j + 1$, otherwise continue.
20. Output the resulting running entropy values as W.

The Python function of the optimized running window entropy is as follows, and is available at (Jones, 2017a, 2017b):

```
01.# Optimized Window Entropy Function
02.def optimized_window_entropy():
03.    # Start the time counter
04.    starttime = time.process_time()
05.    # Entropy list
06.    H = [0 for i in range(1, k - j + 2)]
07.    # Create the count list
08.    m = [0 for i in range(0, 256)]
09.    # Initial population of count list
10.    for x in range(1, j + 1):
11.        m[malware[x - 1]] += 1
12.    # Compute initial entropy
13.    entropy = float(0)
14.    for y in range(0, 256):
15.        if m[y] != 0:
16.            entropy += -(float(m[y] / j) *
   math.log2(float(m[y] / j)))
17.    H[0] = entropy / K
18.    for x in range(2, k - j + 2):
19.        lastval = malware[x - 2]
20.        nextval = malware[x + j - 2]
21.        entropy += float(m[lastval] / j) *
   math.log2(float(m[lastval] / j))
22.        m[lastval] -= 1
23.        if m[lastval] != 0:
24.            entropy += -(float(m[lastval] / j) *
   math.log2(float(m[lastval] / j)))
25.        if m[nextval] != 0:
26.       entropy += float(m[nextval] / j) *
   math.log2(float(m[nextval] / j))
27.        m[nextval] += 1
28.        entropy += -(float(m[nextval] / j) *
   math.log2(float(m[nextval] / j)))
29.        H[x - 1] = entropy / K
30.   # End the time counter
31.   endtime = time.process_time()
32.   return endtime - starttime, H
```

A theoretical algorithm analysis of the source code above leads to a theoretical tilde asymptotic growth of a factor of k*4 which, in theory, is less than $k * j$ when j is greater than 4. In this research j is greater than or equal to 256, so the algorithm is optimized in theory.

Using the evaluation criteria described previously, the average execution times for corresponding experiments as the original algorithm were collected for the optimized algorithm and provided in Table 3, for the empirical measurement.

The data from Table 3 was plotted in a two-dimensional graph in Figure 13. Time is clearly dependent on the size of the malware and less dependent on the size of the running window, as seen in the original algorithm. This supports the theoretical tilde analysis of the

Table 3. The optimized RWE algorithm execution time in seconds

Malware Size (bytes)	256 Byte Entropy Window	512 Byte Entropy Window	1,024 Byte Entropy Window	2,048 Byte Entropy Window
256	1.54E-04	N/A	N/A	N/A
512	8.14E-04	2.45E-04	N/A	N/A
1,024	2.19E-03	1.68E-03	3.20E-04	N/A
2,048	5.23E-03	4.74E-03	3.50E-03	4.97E-04
4,096	1.12E-02	1.10E-02	9.81E-03	6.91E-03
8,192	2.30E-02	2.24E-02	2.27E-02	2.02E-02
16,384	4.35E-02	5.09E-02	5.49E-02	4.91E-02
32,768	9.55E-02	1.00E-01	1.06E-01	1.04E-01
65,536	1.99E-01	2.00E-01	2.16E-01	2.13E-01
131,072	3.93E-01	3.95E-01	4.83E-01	4.24E-01

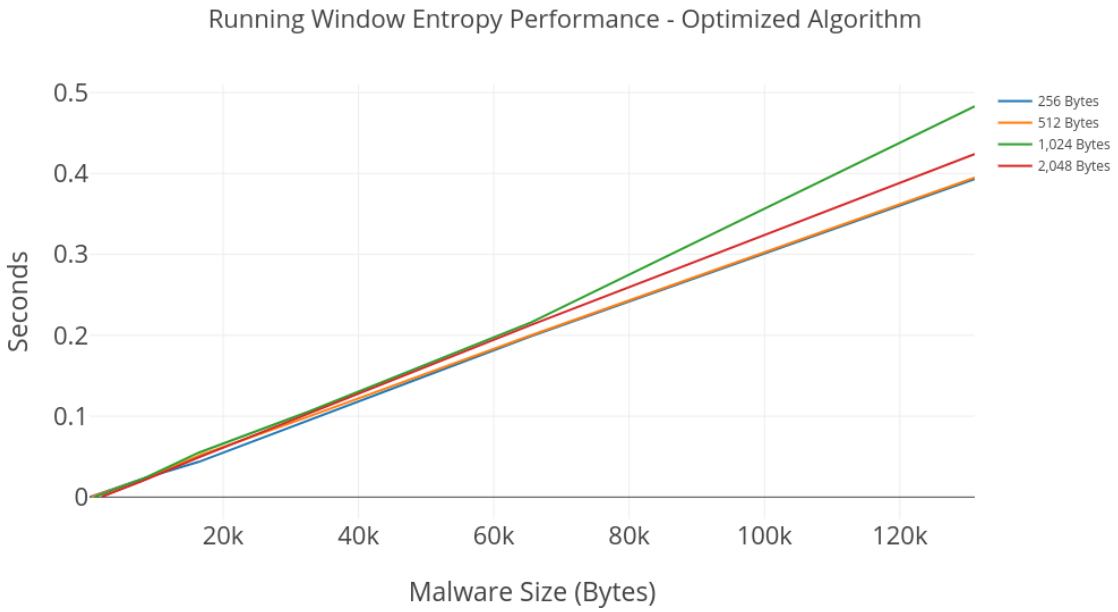


Figure 13. The optimized RWE algorithm performance in seconds

algorithm showing that asymptotic growth of the algorithm is a factor of $k * 4$, the malware size times four.

Evaluation Of The Optimized Algorithm

It is clear from the tables and graphs that the optimized algorithm outperforms the original algorithm theoretically and empirically. Table 4 lists the percentage of time for the optimized algorithm compared to the original algorithm. In most cases, the optimized algorithm requires only 2% or less of the time of the original algorithm. In cases where the window size is equal to the malware size, there is little performance gain. In this situation, the window is unnecessary because the entropy of the entire file is calculated. An inner "for" loop should not be needed in those circumstances. Removing 98% or more of the calculation time with the optimized algorithm proves that this research was successful as defined in the evaluation criteria described previously.

For one average example, the original algorithm requires 23.7 seconds to calculate a 256-byte running window entropy over a 128k byte file, which are modest numbers for real world scenarios. The optimized algorithm requires 0.393 seconds for the same calculation. As discussed earlier, an organization facing 200,000 new malware variants each day, assuming each malware is at least 128k bytes long with a 256-byte running entropy window, the optimized algorithm could save nearly 54 days of computation time alone. At 0.393 seconds per sample, the optimized algorithm could compute over 200,000 samples within the same day. Bigger gains could be realized for parallel and cloud computing.

The correctness of the optimized algorithm was verified against the original algorithm using source code this author developed and available at (Jones, 2017a). Other than slight rounding errors due to fixed numerical sizes in Python, the optimized algorithm produced the same results as the original algorithm.

Table 4. The optimized v. original percentage of execution time for RWE

Malware Size (bytes)	256 Byte Entropy Window	512 Byte Entropy Window	1,024 Byte Entropy Window	2,048 Byte Entropy Window
256	93.55%	N/A	N/A	N/A
512	1.99%	97.61%	N/A	N/A
1,024	1.74%	1.38%	89.77%	N/A
2,048	1.79%	1.26%	0.93%	87.23%
4,096	1.76%	1.24%	0.87%	0.58%
8,192	1.65%	1.14%	0.85%	0.56%
16,384	1.58%	1.24%	0.87%	0.56%
32,768	1.69%	1.20%	0.86%	0.57%
65,536	1.67%	1.18%	0.87%	0.55%
131,072	1.66%	1.16%	0.92%	0.54%

Summary

This chapter presented an independently verifiable, non-trivial optimization technique to reduce the computation of a running window entropy to less than 2% of the original algorithm. This is a method design science artifact. This time savings represents days and months of computation time in conservative scenarios. This algorithm is also used in prior research and applying this optimization will increase performance of prior research. The optimized algorithm presents an incredible time and cost savings to identify relevant portions of malware for the analyst's or security product's attention, while the correctness of the optimized algorithm has been verified against the original algorithm. The source code used in this chapter is publicly available (Jones, 2017b, 2017a) for further research and implementation.

CHAPTER 5

QUALITATIVE DATA EXPLORATION

Introduction

When deciding on a focused topic for this dissertation, a qualitative data exploration phase was completed with approximately 25,190 publicly available malware samples collected from VirusTotal ("VirusTotal," n.d.-a). The exploratory data set was acquired using a series of Yara (VirusTotal, n.d.-b) signatures that were presented in chapter 3. Once the samples were collected, the reported classifications by each vendor were analyzed in order to map several different classifications into the six main functional groups that are the subject of this study. The six functional classifications developed from the exploratory data set will be used to capture the final malware data set used throughout the remainder of this dissertation. Therefore, the data exploration phase, while short, helped improve the overall quality of the final malware classifiers and guided many of the decisions in the research methodology. Without this phase, the research would not have been as successful.

Collecting The Exploratory Data Set

The exploratory data set was sourced quickly in order to begin development on the Malgazer Python libraries at the start of this research. The purpose of the exploratory data set was to use it as example data to create a reusable set of Python libraries. As such, the samples were collected based solely on the file type equaling a Microsoft Windows PE file and at least twenty-five detections (as reported by VirusTotal) using the Yara signature presented in the previous chapter. The Yara signature was allowed to run for some time, and the results were downloaded using the Malgazer Python libraries. The corresponding file content for each malware sample in the VirusTotal feed was downloaded as well.

Computing The Running Window Entropy

After the samples were downloaded, the running window entropy values were calculated for each file. The RWE window size was 256 bytes, while many sampled data points were used ranging from 1,024 to 8,192 bytes. The data was saved to CSV and Pickle formats so they could be used in the future.

Exploring The Classifications

The data returned from VirusTotal can be visualized as a large spreadsheet where the rows represent one malware sample and the columns represent attributes about the samples. The columns returned from VirusTotal in early 2018 were:

- ALYac
- AVG
- AVware
- Ad-Aware
- AegisLab
- AhnLab-V3
- Alibaba
- Antiy-AVL
- Arcabit
- Avast
- Avast-Mobile
- Avira
- Baidu
- BitDefender
- Bkav
- CAT-QuickHeal
- CMC
- ClamAV
- Comodo
- CrowdStrike
- Cybereason
- Cylance
- Cyren
- DrWeb
- ESET-NOD32
- Emsisoft
- Endgame
- F-Prot
- F-Secure
- Fortinet
- GData
- Ikarus
- Invincea
- Jiangmin
- K7AntiVirus
- K7GW
- Kaspersky
- Kingsoft
- MAX
- Malwarebyts
- McAfee
- McAfee-GW-Edition
- MicroWorld-eScan
- Microsoft
- NANO-Antivirus
- Paloalto
- Panda
- Qihoo-360
- Rising
- SUPERAntiSpyware
- SentinelOne
- Sophos
- Symantec
- Tencent
- TheHacker
- TotalDefense
- TrendMicro

- TrendMicro-HouseCall
- VBA32
- 'VIPRE'
- 'ViRobot'
- Webroot
- WhiteArmor
- Yandex
- Zillya
- ZoneAlarm
- Zoner
- date
- eGambit
- first_seen
- id
- last_seen
- match
- md5
- nProtect
- positives
- ruleset_name
- scans
- sha1
- sha256
- size
- subject
- total
- type

The “positives” column provided the number of VirusTotal detections. This column will contain a number equal to or greater than twenty-five because the results were filtered while downloaded using the Malgazer source code. The “sha256” column contained the hash for the malware sample. The SHA256 hash is a computationally secure, mathematically computed value from the file’s contents. This is the unique identifier used to track the samples in this research. Information about any of the samples throughout this research can be searched by anyone on VirusTotal if they have the SHA256 hash.

The data in this spreadsheet was analyzed by visually scanning classifications in each column to find the columns with the most detections that appeared to be consistent. The columns “Microsoft”, “Symantec”, “Kaspersky”, “Sophos”, “TrendMicro”, and “ClamAV” contained data that appeared to be useful for functional classifications, in that order. An example of the data in the spreadsheet is available in Table 5 for eleven malware samples.

It is quickly apparent that not all vendors were able to detect the same samples, therefore multiple vendors were chosen. It is also quickly apparent that detections between vendors were not consistent. For example, the third sample in Table 5 the “Microsoft” column reads “Virus:Win32/Shodi.I” while “Sophos” calls the sample “Mal/Generic-S”. In this case, it would make sense to classify this sample as “Virus” from the Microsoft classification. This is how the final classifications were determined for this research.

Table 5. Example VirusTotal Detections

SHA256	Microsoft	Symantec	Kaspersky	Sophos	TrendMicro	ClamAV
4009b81841c893eceaf2d92e1816a968fad66db69f70429724e6d6f7bb9a4ca7		Trojan.Gen.2			HT_DYNAMER_GA25051D.UVPM	Win.Trojan.Agent-1149280
79b155f341a6c3311b4ded9fc2c239f0e124b705b627cfe62a9b6b6620d80c5b		Trojan.Gen.2		Mal/Generic-S	HT_DYNAMER_GA25051D.UVPM	Win.Trojan.Agent-1149280
eea41fa0f53c20ab2302fdfef59dd4b3dba80f14ec419f9d9c0fb18eac484c1a	Virus:Win32/Shodi.I	Trojan.Gen.NPE.2	Virus.Win32.Virut.ce	W32/Scribble-B	TROJ_OBFUSCATOR_GC140262.UVPM	Win.Trojan.Shohdi-6136104-0
60dcb577124f01cbfcadbe7a86b61d06ad3783083a04f4847a2a255f210ffb28		SMG.Heur!gen	HEUR:Trojan.Win32.StartPage	Cloud Wrapper (PUA)		Win.Trojan.Fam-6454574-1
fbee9ad67e71ac0e109503a2acd0dd7be1e5f91c473bcb349fbbd5e0050a1d3f	Trojan:Win32/Qhost		HEUR:Trojan.Win32.Generic		HT_AGENT_HC050083.UVPM	Win.Trojan.Qhost-160
472257d7fc92f035a50b37ec2536b24bf83ec66ef356df4c03afd8f297138bcf	Virus:Win32/Shodi.I	Trojan.Gen.NPE.2	Virus.Win32.Virut.ce	W32/Shodi-I	TROJ_OBFUSCATOR_GC140262.UVPM	Win.Trojan.Shohdi-6136104-0
f308ad39249c32127653c45e20eb318aca5db848494812d07b688573c128f90a	Trojan:Win32/Dorv.A	Trojan Horse	Trojan.Win32.Inject.azgw	Troj/Simbot-J	TROJ_KRYPTK.SMS	Win.Trojan.Injector-6297684-0
2c1b5dabfc0d3646d27814e8e0092e856debbb704a53de402a9b49ed3a81130a		Trojan.Gen.2		Mal/Generic-S	HT_DYNAMER_GA25051D.UVPM	Win.Trojan.Agent-1149280
5c9888db9bf4aebe7102dc086505aed8afc33ccc7e4bb75257e8052f77f77061	TrojanDropper:Win32/Dinwod.B!bit	Suspicious.IRCBot	HEUR:Trojan.Win32.Generic		HT_AGENT_GL26002B.UVPM	
6d38be3380671eaa11f1dec1758012b6a2e7b8f48ced5bd4edc37ea48c640da4		Trojan.Gen.2	not-a-virus:RiskTool.Win32.FlyStudio.bjxf	Generic PUA JE (PUA)		
b897a0fb1865e016bb88806300ddd9ee739f7919442f40d1b5d22cc974923996	Worm:Win32/Ludbaruma.A	Trojan.Gen.2	Trojan-Ransom.Win32.Blocker.kpuo	W32/Mato-N	TSPY_LUDBARUMA_BK083EDB.TOMC	Win.Worm.Untukmu-5949608-0

In the Malgazer source code, the classification logic can be found in the “Utils” class inside the “estimate_vt_classifications_from_DataFrame” function. There are other helper functions that can load the data from a CSV or HDF file into a DataFrame in the same class as well. This function calls another function named “parse_vt_classifications”, which contains the logic for how each vendor’s classification would be parsed for functional characteristics. Each vendor’s classification was analyzed during this exploratory phase and a parser for each was written. Some of the vendors’ classifications could be parsed with ease, generically, while many other vendors’ classifications required specific logic for their classification. The parsers for all of the vendors can be found in this Python class, which may be useful to others for future research.

Recall from the research methodology’s explanation of the classification determination algorithm that the vendors’ classifications will be searched for certain strings, and the final classification will be determined from a match. The translation table in Table 1 was instantiated in Python code and can be found in the same class. The following code is the translation table as a Python dictionary:

```
01.     translate_classifications = {
02.         'Trj': 'Trojan',
03.         'PUP': 'PUA',
04.         'Adw': 'Adware',
05.         'Drp': 'Dropper',
06.         'Wrm': 'Worm',
07.         'Bkdr': 'Backdoor',
08.         'Cryp': 'Ransom',
09.         'PUA': 'PUA',
10.         'RiskTool': 'PUA',
11.         'Generic': 'Generic',
12.         'VirTool': 'Virus',
13.         'VirLock': 'Ransom',
14.         'HackTool': 'PUA',
15.         'RemoteAdmin': 'PUA',
16.         'Trojan': 'Trojan',
17.         'Mal': 'Generic',
18.         'Malware': 'Generic',
19.         'Worm': 'Worm',
20.         'Troj': 'Trojan',
21.         'FakeAV': 'FakeAV',
22.         'DwnLdr': 'Downloader',
```

```
23.         'TrojanDownloader': 'Downloader',
24.     }
```

The output of the translation will contain more than the six functional groups discussed in this research, since it only translates terms. For this research, just the six functional groups were selected going forward, so only rows containing the six functional names were included. The algorithms written in the Malgazer source code were used to automatically determine this final functional classification of each sample so further analysis could be conducted.

Exploring The New Classifications

Once the new functional classifications were determined, the samples were counted. The "Trojan" classification contained 8,963 samples. The "Worm" classification contained 5,475 samples. The "Virus" classification contained 5,349 samples. The "PUA" classification contained 1,717 samples. The "Backdoor" classification contained 1,708 samples. Lastly, the "Ransom" classification contained just 163 samples. The total number of samples explored in this phase was 23,375.

Note that the classifications are unbalanced. The "Trojan" category is overrepresented with 8,963 samples while the "Ransom" category is underrepresented with only 163 samples. If this data set were to be used for actual classifier training, the classifier would "learn" that most samples were similar to the "Trojan" category while there were not many "Ransom" samples to provide a significant population within the classifier. This problem was solved by selecting a final training data set that is balanced across classifications. The method to remedy this situation will be discussed in a later section.

Experimenting With Classifiers

Once the classifications and running window entropy data were computed for the exploratory data set, several ad-hoc artificial and convolutional neural networks were trained from the data through experimentation. While the actual accuracies were not important as this was not the final data set, it was relevant that running window entropy features produced 80% or better classification accuracy. This accuracy came from classifiers that were not finely tuned, so it was possible accuracies could have been higher. At was at this point the discovery that

running window entropy features for malware classification purposes would be successful. This accuracy was much greater than the random guessing technique with probability of 1/6, so the null hypothesis was not true.

Collecting The Final Malware Data Set

Now that all the parameters of the research were set, it was possible to source the final malware data set. This data set was sourced from VirusTotal as well, using the same basic techniques to source the exploratory data set. Instead of one Yara signature to pull malware, several Yara signatures were created that targeted the six functional groups, mainly through the "Microsoft" classifications, but in some cases all classifications. The Yara signatures used to pull the final malware data set were presented in the research methodology chapter and will not be reproduced here.

Summary

This chapter briefly presented the qualitative data exploration phrase from this research. The chapter began with an explanation of how the exploratory data set was collected from VirusTotal. Next, this chapter discussed the running window entropy feature calculations of the same data set. Then, this chapter discussed the qualitative analysis of the classifications retrieved from VirusTotal, and how the six functional classifications were selected. Once the final classification scheme was decided, statistical information from the exploratory data set was computed. This led to the realization that a balanced final training data set was required. However, some artificial and convolutional neural networks were still trained on the unbalanced exploratory data set and accuracies were above 80%. This provided evidence that running window entropy could be used to functionally classify malware over random guessing. Lastly, the final malware data set was collected. The results of the classifiers trained on the final data set will be presented in detail in the next chapter.

CHAPTER 6

MALGAZER: AN RWE-BASED MALWARE CLASSIFIER

Introduction

This chapter will present the results of developing the Malgazer classifier along with the results of the prior GIST classifier methodology on the same data set. Together, this required over 200 experiments that culminated in months of computation time. Each experiment was a grid search, a cross fold validation, or final training for each of the selected classifier algorithms, hyper-parameters, and input features.

This chapter will begin with a presentation of the grid searching results and the computed hyper-parameters. Next, this chapter will present the results of the ten-fold cross validation accuracy mean and variance using the optimal hyper-parameters. The reason for cross validation testing is to understand how the classifier would perform across the whole data set on average rather than one random selection. Then, this chapter will present the final training results. The Malgazer and GIST classifiers will be theoretically compared to other prior literature in the next chapter.

Computing The Machine Learning Features

After the final data set was collected, the running window entropy features were calculated. Various window sizes and data points were calculated. The window sizes were 256; 512; 1,024; and 2,048 bytes while the data points were 512; 1,024; 2,048; and 4,096 bytes. Additionally, the 320 GIST features were calculated for each sample and stored. The calculations were computed using the “extract_rwe_features.py” and “extra_gist_features.py” scripts in the Malgazer source code repository. These scripts crawled directories to locate files, they calculated the RWE or GIST features, and then they stored the data. The calculated

feature data is publicly available via instructions located in the Malgazer source code repository (Jones, 2017b). Note that it is impossible to recreate the malware samples from the RWE or GIST data because information is lost during the calculations. Therefore, downloading the data will not harm a computer system or proliferate more malware samples.

Scaling The Computations

Once the running window entropy and GIST data were computed from the final malware data set, the training began. Since this research included a larger data set and comprehensively searched for the best model parameters to optimize classification accuracy, much more computing power than this author owned was needed. The following training phases described in the next sections were computed on an assortment of Microsoft Azure and Amazon AWS cloud services. Since this research was conducted by one individual, the 200+ experiments had to be organized and computed as fast as possible, and the application Terraform ("Terraform.io," n.d.) facilitated the creation and destruction of cloud resources on demand.

Terraform is an application that implements infrastructure as code. This means that the cloud computing infrastructure can be declared in Terraform's language and with one command line the services are created or destroyed. This also means that infrastructure can be versioned, just like source code, because the infrastructure is instantiated through source code. The Terraform source code used for Malgazer training research is publicly available at (Jones, 2018). This source code is one of the design science artifacts discussed in the introduction chapter. Although the basics of Terraform is not presented here, one fluent with Terraform could create the same training instances on their Azure or AWS account with the repository above just as this author did. More information is available in the appendix as a source code repository tour.

Searching For The Best Model Parameters

The algorithm for calculating the best model parameters is known as "grid searching" ("Scikit Learn GridSearchCV," n.d.). Several grid searches were performed to find the optimal hyper-parameters of the machine learning algorithms selected for Malgazer. Each

sub-section below will list the tested classifier parameters along with the calculated best parameters for each of the machine learning models using the final training data set. Each of the machine learning models was evaluated using the Malgazer RWE and the GIST features, for empirical comparisons. The optimal hyper-parameters were calculated using the "train_classifier.py" script available in the Malgazer source code repository (Jones, 2017b). The complete logs for each of the training sessions are also available in the same repository under the "training" directory.

k-Nearest Neighbors

The KNN algorithm was used by Method #14 in (L Nataraj et al., 2011), also referred to as the GIST method in this dissertation. In this method, the authors used KNN with Euclidean distance as their machine learning model. Scikit Learn makes this algorithm available in Python ("Scikit Learn KNeighborsClassifier," n.d.). Table 6 and Table 7 present the grid searching results of the KNN machine learning model on the RWE and GIST data sets. For both data sets, using weights as "uniform" with the single closest neighbor provided the best results.

Table 6. Grid search results for KNN and GIST

Machine Learning Model Type:	**k-Nearest Neighbors (KNN)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"n_neighbors":	1, 5, 10, 50
"weights":	"uniform", "distance"
Best Parameters:	**"n_neighbors" = 1, "weights" = "uniform"**
Best Score:	**92.84%**

Table 7. Grid search results for KNN and RWE

Machine Learning Model Type:	**k-Nearest Neighbors (KNN)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"n_neighbors":	1, 5, 10, 50
"weights":	"uniform", "distance"
Best Parameters:	**"n_neighbors" = 1, "weights" = "uniform"**
Best Score:	**88.77%**

Decision Trees

Decision trees are one of the simplest and fastest machine learning model algorithms to implement ("Scikit Learn DecisionTreeClassifier," n.d.). A grid search using decision trees was computed for the RWE and GIST features. The results of the grid search are available in Table 8 and Table 9. Only the "criterion" and "splitter" values can be searched for decision trees. Both of the models achieved the best scores with the "entropy" criteria and a random splitter.

Random Forests

Random forests are a collection of decision trees, and a forest's hyper-parameters include the hyper-parameters from the decision tree model, plus it has the number of estimators (the number of decision trees) and an "oob_score" ("Scikit Learn RandomForestClassifier," n.d.). The model was grid searched for the RWE and GIST

training data and the results are presented in Table 10 and Table 11. Both of the data sets performed the best when the criterion was “entropy”, the number of estimators was five, and the “oob_score” is true.

Nearest Centroid

The nearest centroid algorithm has been implemented by Scikit Learn (“Scikit Learn NearestCentroid,” n.d.). Only one hyper-parameter is available for tuning, the shrink threshold. The nearest centroid algorithm was grid searched for the GIST and Malgazer RWE data, and the best parameters are presented in Table 12 and Table 13.

Support Vector Machines

The support vector machine algorithm has been implemented by Scikit Learn (“Scikit Learn SVC,” n.d.). Several hyper-parameters were grid searched. The best parameters are presented in Table 14 and Table 15. Note that the SVM algorithm is computationally intensive and there are a number of hyper-parameters that can be tuned, which multiplies the number of classifiers that must be tested during the grid search. Therefore, only 20% of the final training data set was used to compute the best parameters for each data set so that the computations were manageable.

AdaBoost And Decision Trees

Adaboost is a machine learning algorithm that improves a base estimator’s accuracy. The Adaboost algorithm has been implemented by Scikit Learn (“Scikit Learn AdaBoostClassifier,” n.d.). Adaboost was grid searched for a decision tree base estimator using the same parameters as the decision tree subsection. The results of the grid search for this machine learning model combination are available in Table 16 and Table 17.

AdaBoost And Random Forests

Adaboost is a machine learning algorithm that improves a base estimator’s accuracy. The Adaboost algorithm has been implemented by Scikit Learn (“Scikit Learn AdaBoostClassifier,” n.d.). Adaboost was grid searched for a random forest base estimator

using the same parameters as the random forest subsection. The results of the grid search for this machine learning model combination are available in Table 16 and Table 17. Note that only 10% of the final training data set was used during the grid search because of the computationally intensive nature. 10% of the final training data set made the computations manageable, for these algorithms.

Table 8. Grid search results for decision tree and GIST

Machine Learning Model Type:	**Decision Tree (DT)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"criterion":	"gini", "entropy"
"splitter":	"best", "random"
Best Parameters:	**"criterion" = "entropy", "splitter" = "random"**
Best Score:	**88.38%**

Table 9. Grid search results for decision tree and RWE

Machine Learning Model Type:	**Decision Tree (DT)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"criterion":	"gini", "entropy"
"splitter":	"best", "random"
Best Parameters:	**"criterion" = "entropy", "splitter" = "best"**
Best Score:	**90.65%**

Table 10. Grid search results for random forest and GIST

Machine Learning Model Type:	**Random Forest (RF)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"criterion":	"gini", "entropy"
"n_estimators":	2, 5, 10, 50, 100
"oob_score":	True, False
Best Parameters:	**"criterion" = "entropy", "n_estimators" = 5, "oob_score" = True**
Best Score:	**87.80%**

Table 11. Grid search results for random forest and RWE

Machine Learning Model Type:	**Random Forest (RF)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"criterion":	"gini", "entropy"
"n_estimators":	2, 5, 10, 50, 100
"oob_score":	True, False
Best Parameters:	**"criterion" = "entropy", "n_estimators" = 5, "oob_score" = True**
Best Score:	**88.08%**

Table 12. Grid search results for NC and GIST

Machine Learning Model Type:	**Nearest Centroid (NC)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"shrink_threshold":	0, 1, 5, 10, 50
Best Parameters:	**"shrink_threshold" = 1**
Best Score:	**47.90%**

Table 13. Grid search results for NC and RWE

Machine Learning Model Type:	**Nearest Centroid (NC)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"shrink_threshold":	0, 1, 5, 10, 50
Best Parameters:	**"shrink_threshold" = 0**
Best Score:	**52.45%**

Table 14. Grid search results for SVM and GIST

Machine Learning Model Type:	**Support Vector Machines (SVM)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	20%
"kernel":	"rbf"
"C":	0.1; 1; 10; 100; 1,000; 10,000; 100,000
Best Parameters:	**"C" = 1,000; "kernel" = "rbf"**
Best Score:	**88.52%**

Table 15. Grid search results for SVM and RWE

Machine Learning Model Type:	**Support Vector Machines (SVM)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	20%
"kernel":	"rbf", "poly", "linear", "sigmoid"
"C":	0.1; 1; 10; 100; 1,000; 10,000; 100,000
Best Parameters:	**"C" = 100; "kernel" = "rbf"**
Best Score:	**87.85%**

Table 16. Grid search results for AdaBoost/DT and GIST

Machine Learning Model Type:	**AdaBoost**
Base Estimator Type:	**Decision Tree (DT)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"base_estimator__criterion":	"gini", "entropy"
"base_estimator__splitter":	"best", "random"
"algorithm":	"SAMME", "SAMME.R"
"learning_rate":	1, 0.9, 0.5, 0.3, 0.1, 0.01, 0.001
"n_estimators":	10, 25, 50
Best Parameters:	**"algorithm" = "SAMME", "base_estimator__criterion" = "entropy", "base_estimator__splitter" = "best", "learning_rate" = 1, "n_estimators" = 50**
Best Score:	**92.27%**

Table 17. Grid search results for AdaBoost/DT and RWE

Machine Learning Model Type:	**AdaBoost**
Base Estimator Type:	**Decision Tree (DT)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"base_estimator__criterion":	"gini", "entropy"
"base_estimator__splitter":	"best", "random"
"algorithm":	"SAMME", "SAMME.R"
"learning_rate":	1, 0.9, 0.5, 0.3, 0.1, 0.01, 0.001
"n_estimators":	10, 25, 50
Best Parameters:	**"algorithm" = "SAMME", "base_estimator__criterion" = "entropy", "base_estimator__splitter" = "random", "learning_rate" = 0.9, "n_estimators" = 50**
Best Score:	**93.61%**

Table 18. Grid search results for AdaBoost/RF and GIST

Machine Learning Model Type:	**AdaBoost**
Base Estimator Type:	**Random Forest (RF)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	25%
"base_estimator__criterion":	"gini", "entropy"
"base_estimator__splitter":	"best", "random"
"base_estimator__n_estimators":	2, 5, 10, 50, 100
"algorithm":	"SAMME", "SAMME.R"
"learning_rate":	1, 0.9, 0.5, 0.3, 0.1, 0.01, 0.001
"n_estimators":	2, 5, 10, 50
Best Parameters:	**"base_estimator__criterion" = "entropy", "base_estimator__n_estimators' = 100, "learning_rate" = 0.5, "n_estimators" = 50**
Best Score:	**90.26%**

Table 19. Grid search results for AdaBoost/RF and RWE

Machine Learning Model Type:	**AdaBoost**
Base Estimator Type:	**Random Forest (RF)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	10%
"base_estimator__criterion":	"gini", "entropy"
"base_estimator__splitter":	"best", "random"
"base_estimator__n_estimators":	2, 5, 10, 50, 100
"algorithm":	"SAMME", "SAMME.R"
"learning_rate":	1, 0.9, 0.5, 0.3, 0.1, 0.01, 0.001
"n_estimators":	2, 5, 10, 50
Best Parameters:	**"base_estimator__criterion" = "gini", "base_estimator__n_estimators' = 100, "learning_rate" = 0.01, "n_estimators" = 50**
Best Score:	**88.32%**

Table 20. Grid search results for OneVRest/DT and GIST

Machine Learning Model Type:	**OneVRest**
Base Estimator Type:	**Decision Trees (DT)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"estimator__criterion":	"gini", "entropy"
"estimator__splitter":	"best", "random"
Best Parameters:	**"estimator__criterion" = "entropy", "estimator__splitter" = "random"**
Best Score:	**84.37%**

Table 21. Grid search results for OneVRest/DT and RWE

Machine Learning Model Type:	**OneVRest**
Base Estimator Type:	**Decision Trees (DT)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"estimator__criterion":	"gini", "entropy"
"estimator__splitter":	"best", "random"
Best Parameters:	**"estimator__criterion" = "entropy", "estimator__splitter" = "best"**
Best Score:	**86.61%**

OneVRest And Decision Trees

The OneVRest algorithm has been implemented by Scikit Learn ("Scikit Learn OneVsRestClassifier," n.d.). The OneVRest algorithm needs a base estimator. The OneVRest algorithm was grid searched with a decision tree machine learning model and the results are presented in Table 20 and Table 21.

OneVRest And Random Forests

The OneVRest algorithm has been implemented by Scikit Learn ("Scikit Learn OneVsRestClassifier," n.d.). The OneVRest algorithm needs a base estimator. The OneVRest algorithm was grid searched with a random forest machine learning model and the results are presented in Table 22 and Table 23.

Table 22. Grid search results for OneVRest/RF and GIST

Machine Learning Model Type:	**OneVRest**
Base Estimator Type:	**Random Forest (RF)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"estimator__criterion":	"gini", "entropy"
"estimator__n_estimators":	2, 5, 10, 50, 100
"estimator__n_jobs":	-1
"estimator__oob_score":	True, False
Best Parameters:	**"estimator__criterion": "entropy", "estimator__n_estimators": 100, "estimator__n_jobs": -1, "estimator__oob_score": True**
Best Score:	**87.95%**

Table 23. Grid search results for OneVRest/RF and RWE

Machine Learning Model Type:	**OneVRest**
Base Estimator Type:	**Random Forest (RF)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"estimator__criterion":	"gini", "entropy"
"estimator__n_estimators":	2, 5, 10, 50, 100
"estimator__n_jobs":	-1
"estimator__oob_score":	True, False
Best Parameters:	**"estimator__criterion": "entropy", "estimator__n_estimators": 100, "estimator__n_jobs": -1, "estimator__oob_score": True**
Best Score:	**87.12%**

OneVRest And KNN

The OneVRest algorithm has been implemented by Scikit Learn ("Scikit Learn OneVsRestClassifier," n.d.). The OneVRest algorithm needs a base estimator. The OneVRest algorithm was grid searched with a KNN machine learning model and the results are presented in Table 24 and Table 25.

Table 24. Grid search results for OneVRest/KNN and GIST

Machine Learning Model Type:	**OneVRest**
Base Estimator Type:	**k-Nearest Neighbors (KNN)**
Feature Type:	**GIST**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"estimator__n_neighbors":	1, 5, 10, 50
"estimator__weights":	"uniform", "distance"
Best Parameters:	**"estimator__n_neighbors" = 1, "estimator__weights" = "uniform"**
Best Score:	**93.26%**

Table 25. Grid search results for OneVRest/KNN and RWE

Machine Learning Model Type:	**OneVRest**
Base Estimator Type:	**k-Nearest Neighbors (KNN)**
Feature Type:	**RWE (256-byte window/1,024 data pts)**
Number of Cross Fold Validation Groups:	2
Percent of Final Training Data Set:	100%
"estimator__n_neighbors":	1, 5, 10, 50
"estimator__weights":	"uniform", "distance"
Best Parameters:	**"estimator__n_neighbors" = 1, "estimator__weights" = "uniform"**
Best Score:	**88.91%**

Cross 10-Fold Validation Scores

Using the parameters computed in the last section, the Malgazer RWE features and GIST features were trained and measured with a ten-fold cross validation. The cross-fold validation scores were calculated using the "train_classifier.py" script available in the Malgazer source code repository. Each subsection will present the cross-fold validation data from each model type.

k-Nearest Neighbors

The k-Nearest Neighbors algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 26.

Decision Trees

The decision tree algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 27.

Random Forests

The random forest algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 28.

Nearest Centroid

The nearest centroid algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 29.

Support Vector Machine

The support vector machine algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 30.

Naïve Bayes

The Naïve Bayes algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 31.

AdaBoost

The AdaBoost algorithm was measured using 10-fold cross validation for various base models. The results of the measurements are available in Table 32.

OneVRest

The OneVRest algorithm was measured using 10-fold cross validation. The results of the measurements are available in Table 33.

Table 26. KNN cross validation scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
GIST	N/A	N/A	94.24%	0.0069
RWE	256	512	91.03%	0.0055
RWE	256	1,024	91.07%	0.0057
RWE	256	2,048	91.10%	0.0055
RWE	256	4,096	91.05%	0.0056
RWE	512	1,024	91.26%	0.0054
RWE	1,024	1,024	91.37%	0.0056
RWE	2,048	1,024	91.57%	0.0057

Table 27. Decision trees cross validation scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
GIST	N/A	N/A	90.27%	0.0078
RWE	256	1,024	92.25%	0.0059
RWE	256	4,096	92.34%	0.0072
RWE	1,024	1,024	92.82%	0.0073

Table 28. Random forest cross validation scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
GIST	N/A	N/A	90.29%	0.0081
RWE	256	512	89.99%	0.0064
RWE	256	1,024	90.36%	0.0078
RWE	256	2,048	90.51%	0.0081
RWE	256	4,096	90.57%	0.0084
RWE	1,024	1,024	91.28%	0.0084

Table 29. Nearest centroid cross validation scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
GIST	N/A	N/A	47.95%	0.0082
RWE	256	1,024	52.39%	0.0073
RWE	256	4,096	52.50%	0.0074
RWE	1,024	1,024	51.29%	0.0066

Table 30. SVM cross validation scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
GIST	N/A	N/A	93.55%	0.0057
RWE	256	512	93.02%	0.0061
RWE	256	1,024	93.24%	0.0054
RWE	256	4,096	93.37%	0.0059
RWE	1,024	1,024	93.14%	0.0056

Table 31. Naïve Bayes cross validation scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
GIST	N/A	N/A	53.64%	0.0075
RWE	256	1,024	56.19%	0.0046
RWE	256	4,096	56.26%	0.0048
RWE	1,024	1,024	54.01%	0.0047

Table 32. Adaboost cross validation scores

Base Model	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
dt	GIST	N/A	N/A	93.68%	0.0060
dt	RWE	256	1,024	94.63%	0.0042
dt	RWE	256	4,096	94.95%	0.0046
dt	RWE	1,024	1,024	94.96%	0.0047
rf	GIST	N/A	N/A	93.93%	0.0051
rf	RWE	256	1,024	94.69%	0.0043
rf	RWE	256	4,096	94.84%	0.0041
rf	RWE	1,024	1,024	95.00%	0.0045

Table 33. OneVRest cross validation scores

Base Model	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
dt	GIST	N/A	N/A	89.03%	0.0083
dt	RWE	256	1,024	90.15%	0.0090
dt	RWE	256	4,096	90.63%	0.0078
dt	RWE	1,024	1,024	90.98%	0.0070
rf	GIST	N/A	N/A	90.37%	0.0097
rf	RWE	256	1,024	89.68%	0.0091
rf	RWE	256	4,096	90.10%	0.0090
rf	RWE	1,024	1,024	90.94%	0.0094
knn	GIST	N/A	N/A	94.24%	0.0073
knn	RWE	256	1,024	91.09%	0.0055
knn	RWE	256	4,096	91.04%	0.0054
knn	RWE	1,024	1,024	91.36%	0.0055
svm	GIST	N/A	N/A	92.71%	0.0070
svm	RWE	256	1,024	91.71%	0.0069
svm	RWE	256	4,096	92.01%	0.0070
svm	RWE	1,024	1,024	91.63%	0.0067

Artificial Neural Networks

The artificial neural network algorithm was measured with 10-fold cross validation using several network structures on the RWE and GIST data sets. Each of the neural networks was trained using different optimizers built into the scientific Python libraries. To understand how the optimizer plays a role in the overall accuracy, the results are reported in Table 34 in full. The structure of each of the five networks listed in Table 34 will be presented in the next subsections.

Table 34. Artificial neural network cross validation scores

Model	Optimizer	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
ann-1	sgd	GIST	N/A	N/A	93.09%	0.0074

Model	Optimizer	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
ann-1	rmsprop	GIST	N/A	N/A	93.46%	0.0063
ann-1	adagrad	GIST	N/A	N/A	93.52%	0.0060
ann-1	adadelta	GIST	N/A	N/A	93.16%	0.0147
ann-1	adam	GIST	N/A	N/A	93.44%	0.0062
ann-1	adamax	GIST	N/A	N/A	93.42%	0.0063
ann-1	nadam	GIST	N/A	N/A	93.63%	0.0060
ann-1	sgd	RWE	256	512	92.43%	0.0056
ann-1	rmsprop	RWE	256	512	94.04%	0.0073
ann-1	adagrad	RWE	256	512	93.96%	0.0046
ann-1	adadelta	RWE	256	512	93.98%	0.0066
ann-1	adam	RWE	256	512	93.65%	0.0051
ann-1	adamax	RWE	256	512	93.30%	0.0042
ann-1	nadam	RWE	256	512	91.74%	0.0391
ann-2	sgd	GIST	N/A	N/A	92.15%	0.0140
ann-2	rmsprop	GIST	N/A	N/A	93.61%	0.0064
ann-2	adagrad	GIST	N/A	N/A	93.48%	0.0063
ann-2	adadelta	GIST	N/A	N/A	93.48%	0.0071
ann-2	adam	GIST	N/A	N/A	93.64%	0.0062
ann-2	adamax	GIST	N/A	N/A	93.30%	0.0048
ann-2	nadam	GIST	N/A	N/A	93.62%	0.0059
ann-2	sgd	RWE	256	512	92.23%	0.0062
ann-2	rmsprop	RWE	256	512	93.90%	0.0061
ann-2	adagrad	RWE	256	512	93.81%	0.0043
ann-2	adadelta	RWE	256	512	93.95%	0.0037
ann-2	adam	RWE	256	512	93.72%	0.0061
ann-2	adamax	RWE	256	512	93.56%	0.0049
ann-2	nadam	RWE	256	512	93.94%	0.0051
ann-3	sgd	GIST	N/A	N/A	93.26%	0.0071
ann-3	rmsprop	GIST	N/A	N/A	93.67%	0.0044
ann-3	adagrad	GIST	N/A	N/A	93.68%	0.0057
ann-3	adadelta	GIST	N/A	N/A	94.08%	0.0052
ann-3	adam	GIST	N/A	N/A	93.84%	0.0065
ann-3	adamax	GIST	N/A	N/A	93.90%	0.0057
ann-3	nadam	GIST	N/A	N/A	93.81%	0.0063

Model	Optimizer	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
ann-3	sgd	RWE	256	512	92.53%	0.0052
ann-3	rmsprop	RWE	256	512	94.12%	0.0050
ann-3	adagrad	RWE	256	512	94.09%	0.0043
ann-3	adadelta	RWE	256	512	94.10%	0.0051
ann-3	adam	RWE	256	512	94.01%	0.0060
ann-3	adamax	RWE	256	512	93.78%	0.0046
ann-3	nadam	RWE	256	512	94.19%	0.0054
ann-4	sgd	GIST	N/A	N/A	93.49%	0.0090
ann-4	rmsprop	GIST	N/A	N/A	93.85%	0.0057
ann-4	adagrad	GIST	N/A	N/A	93.90%	0.0068
ann-4	adadelta	GIST	N/A	N/A	94.09%	0.0065
ann-4	adam	GIST	N/A	N/A	93.96%	0.0043
ann-4	adamax	GIST	N/A	N/A	94.05%	0.0060
ann-4	nadam	GIST	N/A	N/A	93.09%	0.0235
ann-4	sgd	RWE	256	512	92.79%	0.0063
ann-4	rmsprop	RWE	256	512	93.93%	0.0064
ann-4	adadelta	RWE	256	512	94.29%	0.0052
ann-4	adam	RWE	256	512	94.28%	0.0048
ann-4	adamax	RWE	256	512	94.19%	0.0057
ann-5	sgd	GIST	N/A	N/A	92.80%	0.0086
ann-5	rmsprop	GIST	N/A	N/A	92.13%	0.0215
ann-5	adagrad	GIST	N/A	N/A	93.66%	0.0054
ann-5	adadelta	GIST	N/A	N/A	93.70%	0.0054
ann-5	adam	GIST	N/A	N/A	93.80%	0.0057
ann-5	adamax	GIST	N/A	N/A	93.90%	0.0060
ann-5	nadam	GIST	N/A	N/A	92.88%	0.0118
ann-5	sgd	RWE	256	512	92.56%	0.0053
ann-5	rmsprop	RWE	256	512	73.69%	0.2753
ann-5	adadelta	RWE	256	512	94.38%	0.0050
ann-5	adam	RWE	256	512	94.31%	0.0038
ann-5	adamax	RWE	256	512	94.22%	0.0059
ann-5	nadam	RWE	256	512	85.15%	0.2290

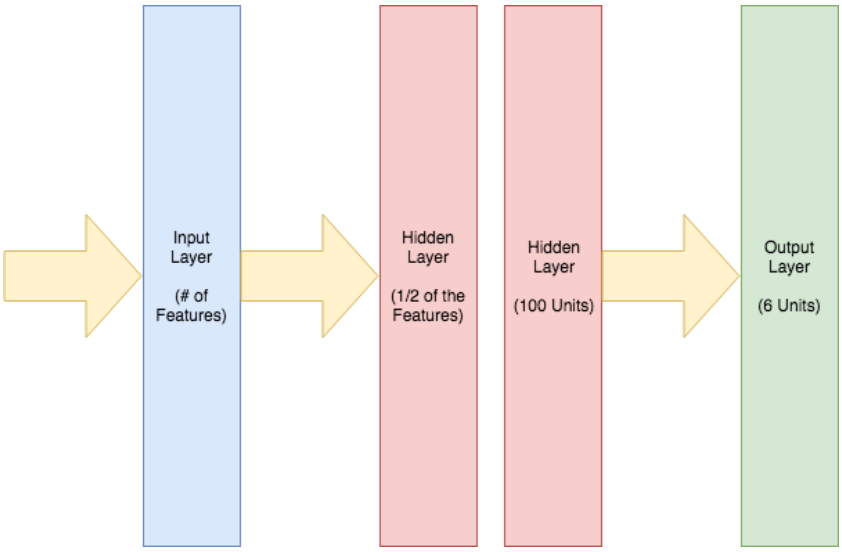


Figure 14. Artificial neural network #1

Artificial Neural Network #1

The artificial neural network number one is presented in Figure 14. The network consists of an input layer with units equaling the number of input features, two hidden layers (with units of ½ the number of features and 100 units), and an output layer with six units, which equals the number of classifications. For the GIST features, there are 320 data points. For the RWE features, the number of data points vary from 215 bytes to 4,096 bytes, depending on the input data set.

Artificial Neural Network #2

The artificial neural network number two is presented in Figure 15. The network consists of an input layer with units equaling the number of input features, four hidden layers (with units of the number of features, ½ the number of features, ¼ the number of features, and 1/8 the number of features), and an output layer with six units, which equals the number of classifications. For the GIST features, there are 320 data points. For the RWE features, the number of data points vary from 215 bytes to 4,096 bytes, depending on the input data set.

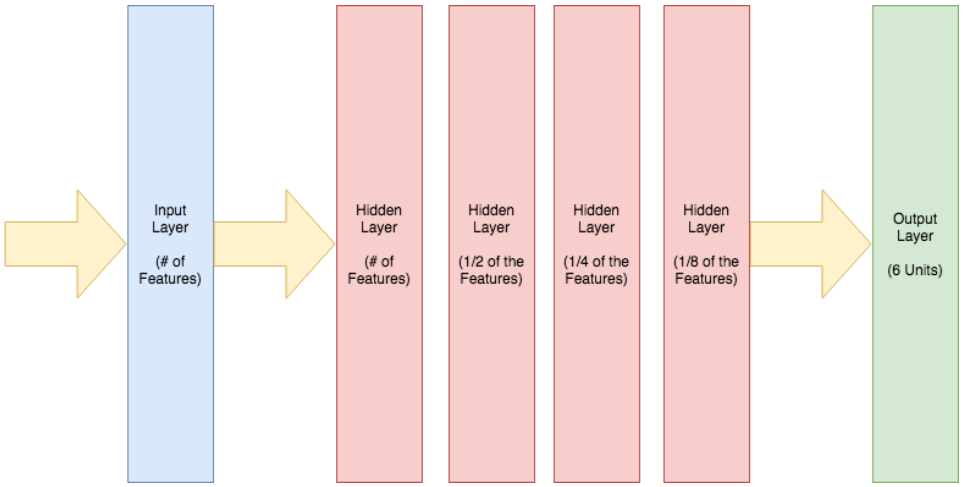


Figure 15. Artificial neural network #2

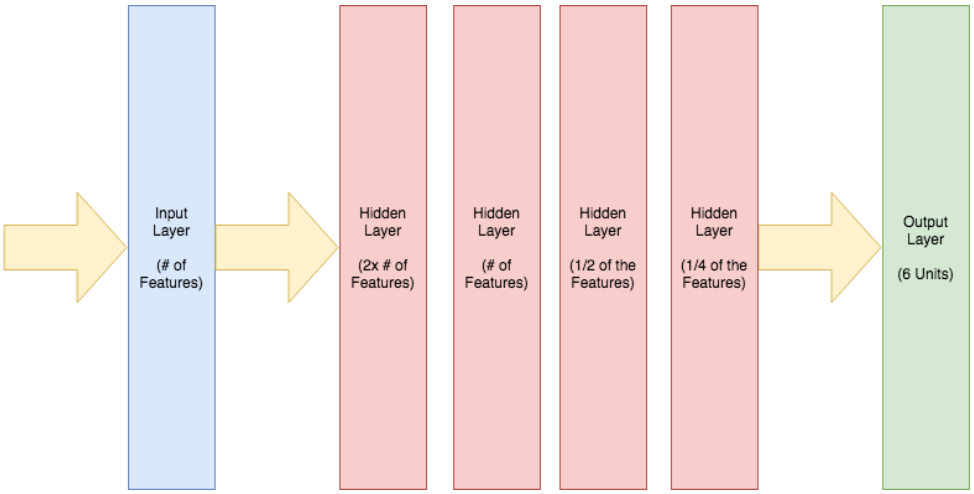


Figure 16. Artificial neural network #3

Artificial Neural Network #3

The artificial neural network number three is presented in Figure 16. The network consists of an input layer with units equaling the number of input features, four hidden layers (with units two times the number of features, the number of features, ½ the number of features, and ¼ the number of features), and an output layer with six units, which equals the number of classifications. For the GIST features, there are 320 data points. For the RWE

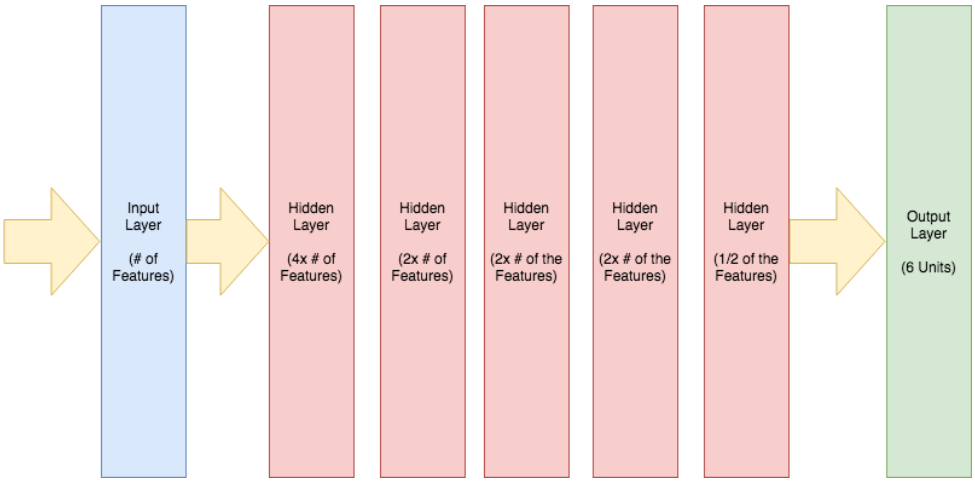


Figure 17. Artificial neural network #4

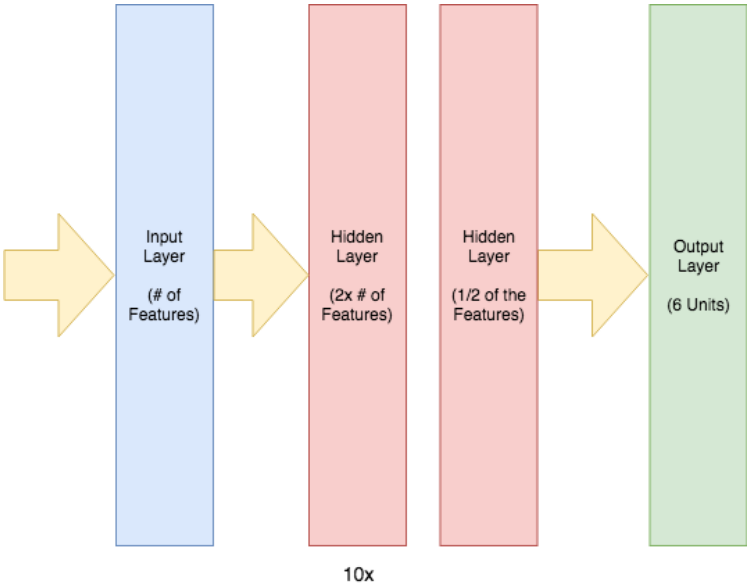


Figure 18. Artificial neural network #5

features, the number of data points vary from 215 bytes to 4,096 bytes, depending on the input data set.

Artificial Neural Network #4

The artificial neural network number four is presented in Figure 17. The network consists of an input layer with units equaling the number of input features, four hidden layers (with units two times the number of features, the number of features, ½ the number of features, and ¼ the number of features), and an output layer with six units, which equals the number of classifications. For the GIST features, there are 320 data points. For the RWE

features, the number of data points vary from 215 bytes to 4,096 bytes, depending on the input data set.

Artificial Neural Network #5

The artificial neural network number five is presented in Figure 18. The network consists of an input layer with units equaling the number of input features, four hidden layers (with units two times the number of features, the number of features, ½ the number of features, and ¼ the number of features), and an output layer with six units, which equals the number of classifications. For the GIST features, there are 320 data points. For the RWE features, the number of data points vary from 215 bytes to 4,096 bytes, depending on the input data set.

Convolutional Neural Networks

The convolutional network algorithm was measured with 10-fold cross validation on the RWE and GIST data sets. The network can be described by the following types of layers:

1. An input layer (the # of features)
2. A one-dimensional convolution with a kernel size of 1/8 of the number of features. There are 100 filters constructed in this layer. The pool size is 10.
3. A one-dimensional convolution with a kernel size of 1/30 of the number of features. There are 100 filters constructed in this layer. The pool size is 2.
4. A one-dimensional convolution with a kernel size of 2. There are 100 filters constructed in this layer. The pool size is 2.
5. A flattening layer.
6. An artificial neural network layer with the number of units ¼ of the number of features.
7. An artificial neural network layer with the number of units 1/8 of the number of features.
8. An artificial neural network layer with the number of units 1/16 of the number of features.
9. An output layer with six values, one representing each class.

The results of the measurements of the CNN are available in Table 35.

Training The Final Classifiers

The Malgazer RWE features and GIST features were trained on several machine learning model types. After the final classifier was trained, the Python object was pickled using the "dill" library so that the classifiers could be reloaded at a later time. Neural networks required multiple files to instantiate the classifier, and the Malgazer classes save or load the networks using supplied functions within the class. Each subsection will present the data from each model type.

k-Nearest Neighbors

The k-nearest neighbors algorithm was trained on the data and measured for accuracy. The results of the measurements are available in Table 36.

Table 35. Convolutional neural network cross validation scores

Model	Base Model	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
cnn-1	rmsprop	GIST	N/A	N/A	92.70%	0.0070
cnn-1	adagrad	GIST	N/A	N/A	90.88%	0.0109
cnn-1	adadelta	GIST	N/A	N/A	92.31%	0.0234
cnn-1	adam	GIST	N/A	N/A	92.40%	0.0078
cnn-1	adamax	GIST	N/A	N/A	91.93%	0.0058
cnn-1	rmsprop	RWE	256	512	93.25%	0.0060
cnn-1	adagrad	RWE	256	512	92.13%	0.0069
cnn-1	adadelta	RWE	256	512	93.47%	0.0061
cnn-1	adam	RWE	256	512	93.11%	0.0067
cnn-1	adamax	RWE	256	512	93.12%	0.0060
cnn-1	nadam	RWE	256	512	93.23%	0.0058

Table 36. KNN final training scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy
GIST	N/A	N/A	94.53%
RWE	256	1,024	91.18%
RWE	256	4,096	91.15%
RWE	1,024	1,024	91.35%

Table 37. Decision trees final training scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy
GIST	N/A	N/A	90.25%
RWE	256	1,024	92.57%
RWE	256	4,096	93.03%
RWE	1,024	1,024	92.73%

Table 38. Random forest final training scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy
GIST	N/A	N/A	90.80%
RWE	256	1,024	90.50%
RWE	256	4,096	91.17%
RWE	1,024	1,024	91.42%

Decision Trees

The decision tree algorithm was trained on the data and measured for accuracy. The results of the measurements are available in Table 37.

Random Forests

The random forest algorithm was trained on the data and measured for accuracy. The results of the measurements are available in Table 38.

Support Vector Machine

The support vector machine algorithm was trained on the data and measured for accuracy. The results of the measurements are available in Table 40.

AdaBoost

The random forest algorithm was trained on the data and measured for accuracy. The results of the measurements are available in Table 39.

OneVRest

The OneVRest algorithm was trained on the data and measured for accuracy. The results of the measurements are available in Table 41.

Table 40. SVM final training scores

Features	RWE Window (Bytes)	RWE Data Points	Accuracy
GIST	N/A	N/A	93.95%
RWE	256	1,024	93.38%
RWE	256	4,096	93.60%
RWE	1,024	1,024	93.05%

Table 39. Adaboost final training scores

Base Model	Features	RWE Window (Bytes)	RWE Data Points	Accuracy
dt	GIST	N/A	N/A	94.03%
dt	RWE	256	1,024	94.97%
dt	RWE	256	4,096	95.32%
dt	RWE	1,024	1,024	95.25%
rf	GIST	N/A	N/A	93.65%
rf	RWE	256	1,024	95.25%
rf	RWE	256	4,096	95.33%
rf	RWE	1,024	1,024	94.90%

Table 41. OnevRest final training scores

Base Model	Features	RWE Window (Bytes)	RWE Data Points	Accuracy
dt	GIST	N/A	N/A	89.10%
dt	RWE	256	1,024	90.33%
dt	RWE	256	4,096	91.12%
dt	RWE	1,024	1,024	91.10%
rf	GIST	N/A	N/A	93.05%
rf	RWE	256	1,024	90.10%
rf	RWE	256	4,096	90.53%
rf	RWE	1,024	1,024	91.27%
knn	GIST	N/A	N/A	94.53%
knn	RWE	256	1,024	91.18%
knn	RWE	256	4,096	91.15%
knn	RWE	1,024	1,024	91.35%
svm	GIST	N/A	N/A	93.05%
svm	RWE	256	1,024	92.18%
svm	RWE	256	4,096	92.35%
svm	RWE	1,024	1,024	91.63%

Artificial Neural Networks

The artificial neural network algorithm was trained on the data and measured for accuracy for the networks previously presented. The results of the measurements are available in Table 42.

Table 42. Artificial and convolutional neural networks final training scores

Model	Features	RWE Window (Bytes)	RWE Datapoints	Accuracy
knn-5	RWE	256	512	94.68%
knn-5	RWE	256	1,024	94.95%
knn-5	GIST	N/A	N/A	93.83%
cnn-1	RWE	256	512	93.88%
cnn-1	RWE	256	1,024	93.62%
cnn-1	RWE	256	4,096	93.47%
cnn-1	GIST	N/A	N/A	92.63%

Convolutional Neural Networks

The convolutional neural network algorithm was trained on the data and measured for accuracy. The results of the measurements are also available in Table 42.

Summary

This concludes the presentation of the results from constructing the Malgazer and GIST classifiers. For each machine learning model selected, the hyper-parameters were grid searched and the optimal hyper-parameters were then measured on the final data set using ten-fold cross validation. Once the optimal classifiers were determined, the final classifiers were saved after training on 90% of the final data set and tested on the remaining 10%. This process, while short when written, consisted of well over 200 time-consuming experiments to collect the data presented here.

The next chapter will compare the Malgazer RWE and GIST results with the prior literature. First, the empirical results will be compared and analyzed between the Malgazer RWE and GIST data sets. Then, a theoretical comparison with other prior literature will be presented to conclude discussion of prior methods.

CHAPTER 7

COMPARISON OF MALGAZER TO PRIOR LITERATURE

Introduction

The previous chapter presented detailed results from over 200 experiments that consisted of grid searching, cross fold validation, and final training. From these results, the Malgazer and GIST classifiers were compared to each other and the prior literature. This chapter begins with the analysis of the empirical results for the Malgazer and GIST classifiers. Next, this chapter compares the Malgazer and GIST classifiers to the prior literature. Even though the input data sets and detailed methodologies were not available to facilitate replication, theoretical comparisons were still drawn from this research to the prior research using the data collected. This chapter concludes with a discussion of other observations learned throughout this research project.

Analysis Of The Empirical Results

This section begins with an analysis of the grid searching results. Next, the cross-fold validation results are analyzed because this data most accurately represented the mean accuracy for the classifiers. Once the cross-fold validation scores have been analyzed, characteristics discovered in the final training experiments are discussed. These three topics are the content of the next three subsections.

Grid Searching Results

The RWE and GIST data sets were used to grid search optimal hyper-parameters for each model. The purpose of this phase was to calculate the optimal hyper-parameters from a set of potential hyper-parameters. Each test consisted of two cross-fold validation groups where the classifier was trained on the first fold and tested on the second fold. For each classifier, a series of potential hyper-parameters were used for training and the best accuracy was reported along

with the optimal set of hyper-parameters. The previous chapter presented the hyper-parameters grid searched in this research, and this section will discuss the results. Note that accuracies were computed during the grid searching phase, but the results should only be used for estimates and general comparisons as the accuracies computed from the cross-validation phase better represented the classifiers' true efficacy.

For KNN, the GIST and RWE optimal hyper-parameters were the same. Not surprisingly, the GIST classifier outperformed the Malgazer RWE classifier. Recall that KNN was the algorithm originally presented in the GIST methodology papers, so the fact that the KNN algorithm performed well for GIST features was expected. Next, the decision tree algorithm was grid searched using the GIST and RWE data sets. The results were same between the data sets, with only the choice of splitter being different ("random" versus "best"). In this test, the RWE classifier outperformed the GIST classifier by over 2%.

Next, the random forest algorithm was grid searched using the GIST and RWE data sets. In this series of experiments, the accuracy and optimal hyper-parameters were the same for both data sets. While the Naïve Bayes algorithm did not have hyper-parameters for tuning, the nearest centroid algorithm did. The GIST nearest centroid algorithm used a shrink threshold of one while the RWE classifier required a shrink threshold of zero. However, for both classifiers the accuracy was below 53%, which well underperformed other classifiers analyzed during this research. In other words, it was wasteful to spend more time on a classifier that was approximately 20% less accurate than other classifiers.

Next, the SVM algorithm was grid searched with the GIST and RWE data sets. For GIST, the optimal hyper-parameter "C" was equal to 1,000. For RWE, "C" was 100. For both data sets, the optimal kernel hyper-parameter was "rbf". The accuracies for both classifiers were within 1%, so the benefits of using GIST versus RWE were more difficult to discern. The SVM algorithm was very computationally intensive and required a long time to complete. Therefore, only 20% of the training data was used to grid search both classifiers. While useful to find the optimal hyper-parameters, it should be noted that grid searching the full data set, if time permitted, may have produced different optimal hyper-parameters and possibly better accuracies than this series of experiments. This was the tradeoff for a shorter computation time for this machine learning algorithm.

The Adaboost algorithm was used to train several base classifiers. The Adaboost algorithm used a series of weak base classifiers to make a larger, more accurate, classifier. First, Adaboost was used with the decision tree base classifier. The hyper-parameters for the GIST and RWE data sets were the same, except for the learning rates. The optimal learning rate for GIST was one, while the learning rate for RWE was 0.9. The RWE classifier outperformed the GIST classifier by more than a percent, therefore the RWE classifier is likely more accurate than the GIST classifier when using Adaboost and decision trees.

Next, Adaboost was used with the random forest base classifier. Like SVM, this training experiment was computationally intensive and only 25% of the data for GIST was used while 10% of the data for RWE was used for training. The optimal hyper-parameters were the same except for the learning rate and base estimator criterion. The base estimator criterion for Adaboost/RF was “entropy”, while the criterion for RWE was “gini”. The learning rate for GIST was 0.5, while the learning rate for RWE was 0.01. It was not fair to compare accuracies between GIST and RWE classifiers from these calculations as different sized training data sets were used. Since the Adaboost algorithm constructed several weak classifiers into a more accurate classifier, other types of base estimators would not work with the Adaboost algorithm. The Malgazer source code repository contains the full set of experiment logs for those interested in exploring more data.

The OneVRest algorithm was used to train several base classifiers. The first base classifier was the decision tree algorithm. The optimal hyper-parameters for OneVRest/DT was “entropy” for the criterion, and “random” or “best” splitters for the GIST and RWE data set respectively. The RWE data outperformed the GIST data by over 2%. Next, random forests were used as the OneVRest base estimator. The grid search determined that the optimal hyper-parameters were the same for GIST and RWE. The reported accuracies were within 1% between the two data sets, making it difficult to declare the most accurate classifier in general. KNN was used as a base classifier for OneVRest next. The optimal hyper-parameters were the same between GIST and RWE, but the accuracy was nearly 5% better for the GIST data set. This was not surprising, given that the original method in the GIST papers used the KNN algorithm.

The artificial and convolutional neural networks were not grid searched in the same manner as the previous algorithms as this too much time and the same information was produced from the cross-fold validation experiments. Recall that the cross-fold validation experiments

provided accuracy mean and variance over ten folds of the data, so this measurement was the best estimate of a classifier's true accuracy. Now that the optimal hyper-parameters were calculated for many machine learning algorithms, the cross-fold validation phase was executed next. The cross-fold validation experiments will be analyzed in the next section.

Best Overall Accuracy From Cross Fold Validation

The results of the cross-fold validations presented in the previous chapter can be sorted by accuracy mean to yield the most performant models. The top twenty-eight models are available in Table 43. It is important to note that most of the models performed very well with the RWE and GIST features, but the Adaboost algorithm with decision trees and random forests trained from the RWE data was the most accurate. The nearest centroid and Naïve Bayes algorithms did not perform nearly as well as the other algorithms, so they will no longer be discussed as there are many more accurate classifiers.

Next, the more complex RWE based artificial neural networks performed better than the GIST methodology. After the GIST methodology, artificial neural networks using both GIST and RWE features performed well. The differences between the most accurate model and the twenty-eighth most accurate model was slightly over 1%, which was clearly an improvement but not a large improvement. This coupled with the fact that the most performant model was only accurate 95% of the time on this particular data set, was noteworthy. As presented in the literature review chapter, most prior literature reported accuracies well above 97%. There are several possible explanations for this accuracy difference. The two or more percent difference could be attributed to the final malware data set rather than the algorithms or types of input features used (i.e. RWE or GIST). The lower accuracy could also be attributed to the selection of the six functional groups. The possibilities for this accuracy difference will be explored later in this dissertation.

Returning to the publication where the GIST method originated (L Nataraj et al., 2011), the authors reported a "98% classification accuracy on a malware database of 9,458 samples with 25 different malware families." The best accuracy for the same classifier trained on the Malgazer data set using the functional classifications developed during this research was 94.24%. This was nearly a 4% difference. There are several possible explanations for this discrepancy. First, this research did not have access to the 9,458 samples used in GIST paper to

validate the reported accuracies in the original paper. It is possible that this lower number of samples did not adequately represent the current real-world threat landscape and could have resulted in inaccurate accuracy scores. Also, the samples could have been too similar to allow for a more realistic comparison to other malware samples not in the training set. It is also possible that the 25 malware families in the GIST training set was more specific than the functional classifications developed in this research.

Table 43. Top 28 classifier accuracies in descending order

Model	Base Model / Optimizer	Features	RWE Window (Bytes)	RWE Data Points	Accuracy Mean	Accuracy Variance
adaboost	rf	RWE	1,024	1,024	95.00%	0.0045
adaboost	dt	RWE	1,024	1,024	94.96%	0.0047
adaboost	dt	RWE	256	4,096	94.95%	0.0046
adaboost	rf	RWE	256	4,096	94.84%	0.0041
adaboost	rf	RWE	256	1,024	94.69%	0.0043
adaboost	dt	RWE	256	1,024	94.63%	0.0042
ann-5	adadelta	RWE	256	512	94.38%	0.0050
ann-5	adam	RWE	256	512	94.31%	0.0038
ann-4	adadelta	RWE	256	512	94.29%	0.0052
ann-4	adam	RWE	256	512	94.28%	0.0048
ovr	knn	GIST	N/A	N/A	94.24%	0.0073
knn	N/A	GIST	N/A	N/A	94.24%	0.0069
ann-5	adamax	RWE	256	512	94.22%	0.0059
ann-3	nadam	RWE	256	512	94.19%	0.0054
ann-4	adamax	RWE	256	512	94.19%	0.0057
ann-3	rmsprop	RWE	256	512	94.12%	0.0050
ann-3	adadelta	RWE	256	512	94.10%	0.0051
ann-3	adagrad	RWE	256	512	94.09%	0.0043
ann-4	adadelta	GIST	N/A	N/A	94.09%	0.0065
ann-3	adadelta	GIST	N/A	N/A	94.08%	0.0052
ann-4	adamax	GIST	N/A	N/A	94.05%	0.0060
ann-1	rmsprop	RWE	256	512	94.04%	0.0073
ann-3	adam	RWE	256	512	94.01%	0.0060
ann-1	adadelta	RWE	256	512	93.98%	0.0066
ann-1	adagrad	RWE	256	512	93.96%	0.0046
ann-4	adam	GIST	N/A	N/A	93.96%	0.0043
ann-2	adadelta	RWE	256	512	93.95%	0.0037
ann-2	nadam	RWE	256	512	93.94%	0.0051

On the other hand, it is possible that the Malgazer training data set did not adequately reflect the functional classifications, and the true accuracy using a different classification strategy was closer to the originally reported accuracy of 98%. Whichever reason might explain the accuracy discrepancies, it is important to note that the Malgazer and GIST classifiers were trained on the same Malgazer dataset with the same known functional classifications applied equally. The differences in accuracies between features could be noted. In other words, if the training data set was not optimal and the true accuracy of the GIST method was actually around 98%, it is possible that the Malgazer classifier would have performed better than 98% using Adaboost and a random forest (or a decision tree) on that training set. This qualitative observation leads from the fact that the most accurate RWE based model was more performant than the most accurate GIST model. In fact, 78.57% of the top 28 classifiers trained during this research were RWE based classifiers. There were four types of machine learning models for RWE that outperformed the best GIST based classifier: adaboost/rf, adaboost/dt, ann-5, and ann-4.

It is now possible to conclude that the training data set was extremely important to the overall functional classification accuracy. If one classification was notoriously low in all cases, the training data set or the classification strategy could have been to blame. The next section analyzes the classification accuracies per class to find potential classifier improvements.

Classification Accuracy Per Class For The Final Training

During the final training phase, the Malgazer source code calculated and plotted the area under the curve (AUC) of the receiver operator characteristic (ROC) using the SciKit Learn functions. The ROC curves were plotted for each model as a whole, plus the classification accuracy for each functional classification was computed and plotted independently. The highest accuracy model (Adaboost/RF) using running window entropy with window size 1,024 bytes and 1,024 data points has been plotted in Figure 19. The “Trojan” classification had the lowest AUC of 0.94, while the Virus classification had an AUC of 0.99. The next highest accuracy model was Adaboost/DT and its ROC curve is presented in Figure 20. The “Trojan” classification once again was the lowest AUC with a value of 0.95, but the “Backdoor”, “Virus”, and “Worm” categories all had the highest AUC of 0.98.

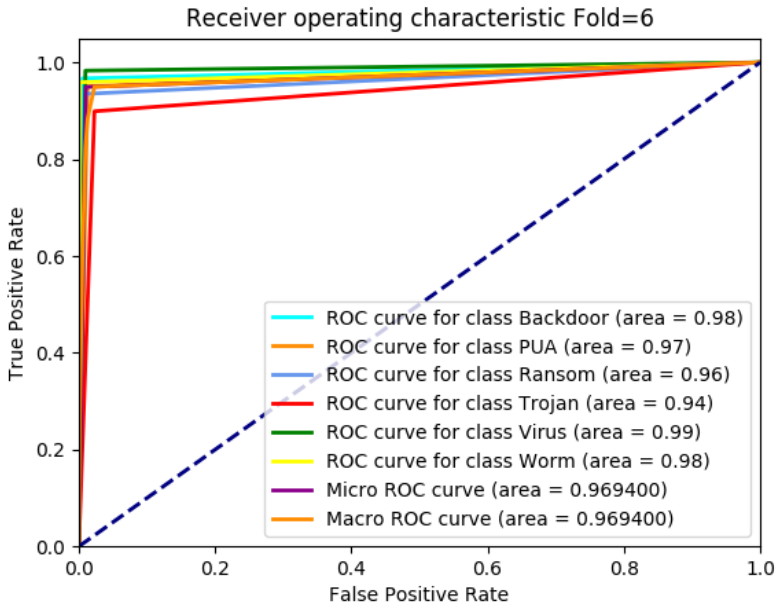


Figure 19. The AUC/ROC for Adaboost/RF (1,024-byte window, 1,024 data points)

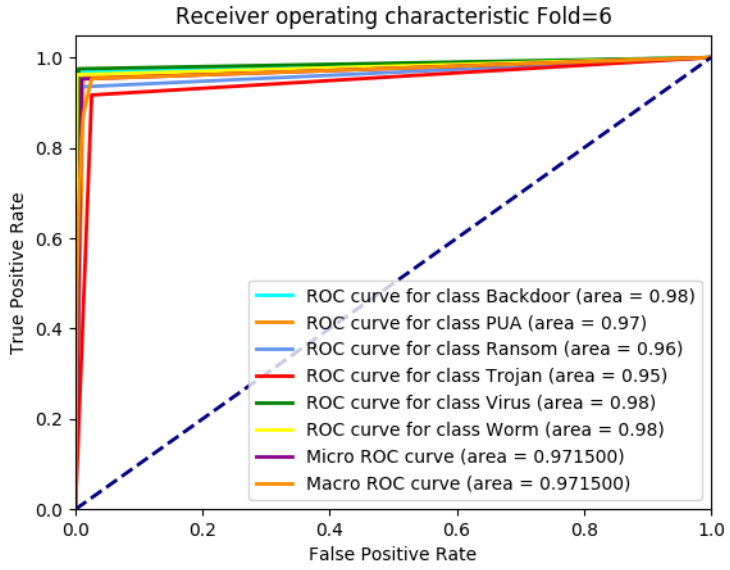


Figure 20. The AUC/ROC for Adaboost/DT (1,024-byte window, 1,024 data points)

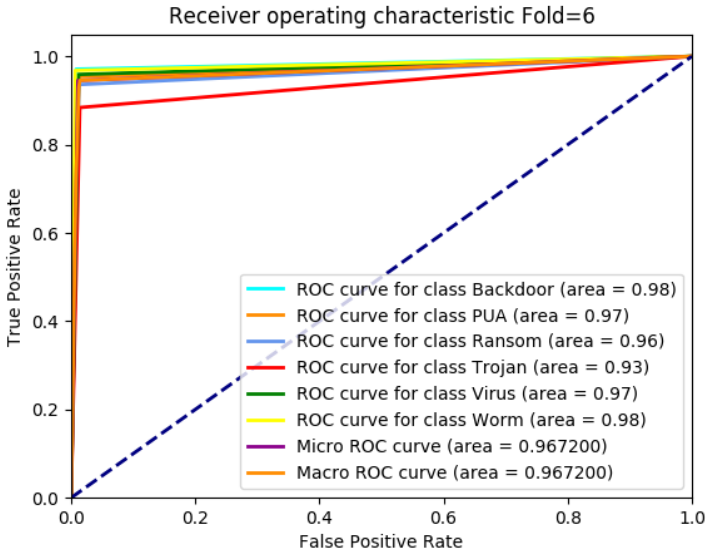


Figure 21. The AUC/ROC for GIST and KNN

The GIST based method with the highest accuracy came from the k-nearest neighbor algorithm. The ROC curves for this model are presented in Figure 21. Once again, the "Trojan" classification received the lowest score with an AUC of 0.93. In every instance of final training, "Trojan" was consistently low. Sometimes "Trojan" was joined with "PUA", "Virus", and "Ransom" as being the lowest correct classification rates for some classifiers.

The fact that "Trojan" was consistently low may have led to lower overall accuracies when compared to prior literature. For example, if the "Trojan" category were removed completely, the accuracies of all experiments would have surely increased because the least accurate classification was removed from consideration. Removing the "Trojan" classification from the model before it was trained would benefit not only the Malgazer RWE classifiers, but it would also benefit the GIST classifiers, according to the ROC curves. This is one method that could be used to improve the classifiers further, if time allowed.

One may wonder why the "Trojan" classification could be problematic and the logic behind removing the classification. To answer this question, an explanation of general malware analysis is required. Malware can have nearly any type of functional characteristics, as long as the sample performed malicious actions. One malware sample may have been ransomware, while another sample may have stolen banking information from a web browser. Sometimes,

one malware sample can have multiple functional characteristics, which made mapping the characteristics into one discreet functional classification difficult.

There was also the avenue of infection that can play a role in the classification scheme. For example, the classification "Trojan" better described the infection mechanism than a functional description. A "Trojan" was an application that looked like a benign piece of software while it still hid a malware payload. A "Trojan" could have also been a "Ransom" sample if the payload locked the user's files for ransom. The same trojan malware payload hiding in two different benign software applications would also have had completely different RWE or GIST values if the benign applications used to hide the malware payload were drastically different. Even ignoring the effects of a malware payload does not reduce the differences in the computed RWE and GIST values if the benign applications were drastically different. The reason the benign applications factor heavily into those calculations is because it wraps a malware payload. These drastic differences in the benign "wrappers" could have affected the classification accuracies for "Trojan".

All of the ROC curves for the final training can be found with the corresponding logs in the Malgazer source code repository (Jones, 2017b). More data exists in the repository than was reported here due to the large number of experiments, but all of the notable experiments were discussed in this dissertation. Those readers interested in more details would benefit from visiting the Malgazer repositories.

Comparing Malgazer To Prior Literature

In order to compare Malgazer to prior literature, it was important to understand the components of Malgazer with respect to the generalized machine learning malware classification model since this model allowed for comparisons between methods at the component level. Based upon the analysis of the training results in this and the prior chapters, Malgazer was categorized as follows:

- CATEGORY: Malware Classifier – Running Window Entropy
- EFFICACY: accuracy of approximately 95%

- TRAINING DATA: 60,000 malware samples and functional classifications from VirusTotal, where there were six functional classifications developed from the publicly available detection information: Virus, Trojan, Worm, PUA, Ransom, and Backdoor
- TRANSFORMATION FUNCTION T(): The RWE data
- PRE-PROCESSING FUNCTION P(): The re-sampled RWE to a fixed sample size
- MACHINE LEARNING MODEL FUNCTION MLM():
    - Adaboost/DT
    - Adaboost/RF
    - Artificial neural networks
- ERROR FUNCTION E(): Confusion matrices
- TUNING FUNCTION TU(): Window and data point sampling sizes for running window entropy.
- POST PROCESSING FUNCTION PP(): Not discussed.

Recall in the literature review chapter that thirty prior methods were malware classifiers. The thirty prior methods were determined by applying the generalized model of malware classification to prior literature and filtering the results so that they contained only malware classifiers (instead of malware detectors, for example). For convenience, the list of the thirty prior methods are reproduced below. Each method is briefly discussed after, and points will be made only once instead of repeating the same point for each method that was obviously similar. Other observations discovered during this analysis are presented in the last section of this chapter.

- Method #13
- Method #14
- Method #15
- Method #18
- Method #19
- Method #25
- Method #28
- Method #29
- Method #31
- Method #33
- Method #34
- Method #35
- Method #36
- Method #38
- Method #40
- Method #41
- Method #44
- Method #45
- Method #50
- Method #51
- Method #54
- Method #56
- Method #57
- Method #58

- Method #59
- Method #60
- Method #61
- Method #66
- Method #67
- Method #71

In Method #13, the authors reported an accuracy of greater than 99%. This was an example that if based strictly on reported accuracies made the comparison with Malgazer and the GIST implementations very difficult. 99% accuracy was very high, and much higher than the accuracies calculated during this research. However, the training data set in this method was less than one third of the training data set used in this research. It is quite possible that the training data set used in this research contained more real-world, and current, samples than the method from eight years prior. Since the GIST authors (Method #14) reported an accuracy of approximately 98% but was measured to be approximately 94% in this research, it is entirely possible that Method #13 would test lower as well. This would leave open the possibility that Malgazer's RWE based classifiers could match or outperform this prior method.

In Method #15, the authors used dynamic analysis information from just over three thousand training samples. The number of samples used in this prior method was only 5% of the number used during this research, so applying the prior method to Malgazer's data set would likely decrease the overall accuracy. Interestingly, since this method used dynamic malware analysis information as the input data, dynamic features such as this could have been added to the Malgazer or GIST classifiers. The features may have worked in parallel with each other in the quest for the most accurate classifier.

In Method #18, no efficacy was reported. This method only used 9,442 malware samples for training, which was less than one sixth of the data examined in this research. In Method #19, the authors used a combination of techniques that led them to image similarity. The authors reported an accuracy of 98.66%. This paper was similar to the GIST paper, in that they used a large number of families as classifications rather than just six functional categories. The accuracy was also similar to the GIST paper of approximately 98%. The training data set was much smaller than in this research, with the authors reporting approximately 2,460 malware samples used for training. Although the originally reported accuracy was higher, it would likely reduce as more samples were added to the training data set. With only 100 total samples per family composing the training data, this classifier would likely become less accurate with the larger data sets used in this research.

In Method #25, the authors used a technique to translate malware samples into gray scale images. The gray scale images were classified using a convolutional neural network. Convolutional neural networks are known for their accuracy when classifying one- or two-dimensional structured data, so it was not surprising that this method reported an accuracy of 98.56%. The authors reported using a data set of 21,741 samples, which was approximately a third of the size used in this research. Furthermore, the data set was classified by family name, and this research examines functional classifications, so accuracy differences could be attributed there as well.

In Method #28, the authors used Markov models to classify a data set of approximately 11,000 samples. The authors reported an AUC of 0.977 which is very close to the AUCs reported in Figure 19 and Figure 20. According to the results, the Malgazer classifier at least matched this prior method for accuracy, even though they classified two different types of classifications (family vs. functional).

In Method #29, the authors used dynamic analysis features to classify a training data set. The authors reported an accuracy of 89.4%, which was lower than the classifiers developed in this research and the GIST classifier. In Method #31, the authors used a very large data set of 6.5 million files to classify malware based upon dynamic analysis. The authors reported an error rate of 2.94%, which would have made the accuracy approximately 97%. This value was not a surprise given the large training data set, as it was in the range considered normal for machine learning based malware detection. This accuracy likely reflected the true accuracy of this method because the size of the training data set was very large.

For both methods discussed in the prior paragraph, dynamic analysis data was required for classification. This means that the samples were executed in a sandbox environment to collect the dynamic data. First, some use cases may not permit the time required for this lengthy process, usually lasting several minutes per sample. Second, dynamic analysis may not have revealed the whole malware payload as only certain code paths are typically executed in those automated environments. The data from a missing a code path could have flipped a sample from one classification to another. This observation could be true for most any purely dynamic analysis feature set.

In Method #33, the authors used a neural network to classify malware into families. The authors reported an accuracy of 96% for approximately 5,000 malware samples. This

classification accuracy was very close to the neural networks studied in this research and was within 1% accuracy of the most performant model. Perhaps the difference in accuracy was due to the classification scheme (functional vs. family) or the small training data set used in the prior research.

In Method #34, the authors proposed a convolutional neural network family name classifier using gray scale images of the malware samples as input. The authors reported an accuracy of 98% using an unbalanced data set. As discussed throughout this dissertation, unbalanced data sets could lead to inaccurate results because some categories would have been more likely than others. Although the reported accuracy for this classifier was within a reasonable range, it may have been less accurate in the real world, where the threat landscape changes frequently.

In Method #35, the authors used numerous static analysis features with a convolutional neural network to classify approximately 22,757 samples. The authors reported an accuracy of 93%, which was lower than the classifiers created in this research. This method used approximately one third of the samples used in this research.

In Method #36, the authors used an intermediate language to represent the malware samples and classify them with a LSTM neural network. The authors reported an accuracy of 97.7% and an AUC of 0.9832, which are higher than the values for the classifiers developed in this research. The authors used approximately 310,000 samples as their data set, which was five times more samples than used during this research. This method also studied an LSTM neural network, which is a different type of neural network that was not studied during this research.

In Method #38, the authors used various machine learning models to classify malware based on static and dynamic features. The authors reported an accuracy of 92%, which was lower than the classifiers developed during this research. The authors also used only 3,320 malware samples, which was only 5.5% of the size of the training set used for this research. In Method #40, the authors reported a 7.2% error rate (approximately 93% accuracy) classifying millions of malware samples. This method used family names for classifications. A 93% classification rate for millions of samples demonstrated that larger data sets generally reduce classification accuracy because there were more outliers than in smaller training sets.

In Method #41, the authors used convolutional neural networks to classify gray scale images of malware samples. The authors reported that with approximately 9,339 samples they

achieved an accuracy of 98.62%. This method used less than one sixth of the samples used in this research. 98% accuracy would be difficult to achieve with much larger data sets. In Method #44, the authors used Markov models to classify approximately 11,000 malware samples with accuracy greater than 90%. This accuracy was lower than most accuracies in the literature review, and lower than the 95% accuracy from Malgazer classifiers.

In Method #45, the authors used gray scale images of malware samples as input to convolutional neural networks. The reported accuracy was 95.66%, which was very close to the accuracy calculated in this research. The authors trained their classifiers on 10,000 benign and 2,000 malware files, so it appeared that the data set was unbalanced. In Method #50, the authors used a convolutional neural network to classify a data set of approximately 5,647 samples from four families. The authors reported an accuracy of 98%, which was within range of the other classifiers examined in this chapter.

In Method #51, the authors classified malware using LSTM neural networks on dynamic analysis features from 75,000 malware samples. The authors report a false positive rate of 1% and an improvement in the true positive rate, but it is unclear how this can be compared directly to the data collected from Malgazer. In Method #54, the authors presented a method to classify malware using dynamic analysis data with a reported accuracy of 99.06%. The training data set for this method was 23,080 collected malware samples. Based upon the reported accuracy alone, the value seemed abnormally high. Analysis of the original training data set used may have led to conclusions as to why the reported accuracy was so high for this method.

In Method #56, the authors presented a method to classify malware using gray scale images and convolutional neural networks. The authors reported an error of 31.34%, which would mean the accuracy was approximately 70%. Based upon the work in this dissertation and other literature in the review, this accuracy seemed lower than it should have been. More analysis into this method would have to be completed in order to compare this research to the prior method any further.

In Method #57, the authors presented a classifier built from 10,869 samples that was 95% accurate. This classifier used family names as the classification. The reported accuracy of 95% was the same as the accuracy calculated in this research. In Method #58, the authors presented a classifier built from gray scale images and static information from malware samples. The authors reported an accuracy of 98.862%. The authors stated that they trained their

classifier with approximately 10,868 malware samples, which was much smaller than the data set in this research.

In Method #59, the author presented a classifier built from the sample's function call graphs as features to a neural network. The author reported an accuracy 97.9%, which was in the range of most reported accuracies. The classifier was built from approximately 10,867 malware samples, which was approximately one sixth the size of the Malgazer data set. In Method #60, the authors presented a classifier built from sampled byte code using convolutional and BiLSTM neural networks. The authors reported an accuracy of 98.2%. This research, like many discussed above, used the Microsoft Kaggle (Ronen, Radu, Feuerstein, Yom-Tov, & Ahmadi, 2018) data set of approximately 10,868 malware samples labeled with family names.

In Method #61, the authors presented a method to classify malware based on gray scale images of the bytecode and opcode features with convolutional and LSTM neural networks. The authors reported a classification accuracy of 99.36%. The classifier was built with approximately 41,000 samples. Without analyzing the training and testing datasets used, it was difficult to discern why this method had such a high accuracy score. In Method #66, the authors presented a classifier with 99.97% accuracy that used gray scale images of the malware sample with a convolutional neural network. The classifier was built with approximately 9,339 malware samples, which made the reported accuracy very high for such a small training data set.

In Method #67, the authors presented a method to classify malware using gray scale images and a convolutional neural network. The authors reported an 81.8% accuracy, which is lower than most other image-based methods discussed in this chapter. In Method #71, the authors presented a classifier that used dynamic analysis features to classify approximately 17,400 malware samples into sixty malware families. The authors reported an accuracy of 98%, which was the accuracy most frequently reported throughout the literature review. The choice of family names and the smaller data set could have contributed to the high accuracy score.

In general, classification accuracies of approximately 98% was most frequently reported. A 98% accuracy was considerably better than the 95% accuracy calculated during this research, especially when an organization would be classifying hundreds of thousands of samples per day and 3% can represent thousands of samples. However, it is very important to remember that the accuracy score was relative to the conditions of the experiments. As noted for a number of the studies above, the choice of training data, classification scheme, implementation, measurements,

or any number of other factors could have played a role in the overall higher classification accuracy reported in prior literature. This observation is supported by the fact that very similar methods in prior literature would produce very different accuracy results, such as the image-based methods ranging from approximately 70% to 99% or more. Therefore, it was just as unfair to blindly compare the reported accuracies between two independent malware classification publications as it would have been to compare two studies for any other science where the exact conditions of each experiment cannot be replicated for both studies. It was for this reason that this author publicly released all of the source code, logs, and data sets used throughout out this research. With the code and data publicly available, improvements can be made to Malgazer's algorithms and their effects can be easily measured by anyone with Python programming knowledge.

Other Observations

During the course of this research project, additional discoveries became apparent as the malware classifiers were developed. The results demonstrated that an RWE based classifier improved malware functional classifications over the GIST method if the Adaboost algorithm was used with a decision tree or random forest base classifier. The other type of classifier where RWE outperformed GIST was with the larger neural network. Despite the benefit of improved accuracy, there were also some potential drawbacks when using RWE that must be known for some use cases.

Since Malgazer's classifiers are built from RWE data, the computations are much slower than the GIST algorithm. This research calculated the RWE data and after it also computed the GIST data from the same samples. The RWE calculations required several days to complete, while the GIST data only required a few hours. The penalty in computation time was a trade-off for a higher number of features. The GIST data produced 320 features per malware sample, while the RWE data size was dependent on the malware sample's size and the running window size. Then, the RWE data was down sampled to a smaller number of data points, ranging from 512 to 4,096 bytes. The result was that the GIST data was much faster to compute but contained less data than RWE. The RWE took longer to compute, but there was much more data to use.

This tradeoff is one the user would need to make for their specific use case. For users that already used entropy during malware analysis, the computation of RWE may simultaneously

be valuable for that analysis and the classification problem. If RWE was computed for other purposes than classification, the data could serve multiple purposes and be used to generate Malgazer classifications as well. Once the running window entropy was calculated for a sample, it will never change, so it only ever needs to be calculated and stored once. On the other hand, if the use case required that the computations executed faster, and that entropy data would not be used anywhere else in the malware analysis pipeline, that user may wish to compute and use the GIST data with a KNN classifier to save time and storage space. The choice of classifier should fit the use case for which it was selected.

Summary

This chapter presented an analysis of the results computed from the last chapter. The chapter began with an analysis of the grid searching results to find the optimal hyper-parameters. While finding the hyper-parameters, the grid searching also provided estimates of each classifier's accuracy. Next, the classifiers were cross-fold validated to find the accuracy mean and variance. This measurement provided the grounds on which to compare the Malgazer RWE and GIST classifiers. Once the RWE and GIST classifiers were compared, the classification accuracy per class was explored and it helped explain a possible reason the RWE and GIST classifiers appeared to have performed worse than the prior literature. Then, the malware classifiers from prior literature were compared theoretically to the Malgazer RWE classifier. This chapter concluded with additional research observations that were noteworthy. For example, factors when choosing a malware classifier were discussed for the different types of use cases. The next chapter will present the web application before presenting the overall conclusions in the last chapter.

CHAPTER 8

DEVELOPING THE MALWARE CLASSIFICATION WEB APPLICATION

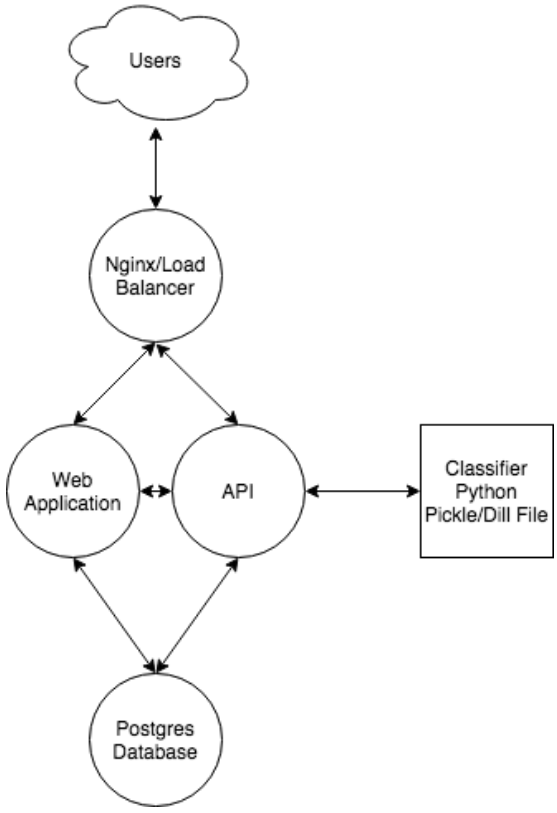
Introduction

Until now, all interactions with any classifier developed throughout this dissertation occurred in the Python interpreter because the classifiers were instantiated objects loaded into memory. To use the classifiers in this state, one would have to be familiar with Python and its interpreter, which is not a trivial requirement. Instead of requiring Python familiarity, this author created the Malgazer web application so that users from all backgrounds can utilize this research. This chapter will briefly discuss the web application architecture, how the application was created, and how to utilize it in order to make full use of this research. Since this design science artifact is an instantiation of a classifier application rather than a new method, the source code will not be presented line by line. Users interested in such information should review the source code in the repository directly for the most detailed implementation information.

Note that this chapter was based upon the "dissertation" git branch in the Malgazer source code repository (Jones, 2017b). This branch was created when this dissertation was published so that the text and the source code could be matched by future readers. It is important to know that the source code in the repository most likely has been improved since this dissertation was published, so interested users should read the "Readme.md" file in the "master" git branch for the most up to date and stable information.

The Malgazer Web Application

The architecture for the web application is presented in Figure 22. At the top, a load balancer routes requests to either the web application or the API, depending on the incoming URL. The web application and API share data through the Postgres database backend. The



classifier's Python pickle/dill files are loaded by the API so that they can be used to make classifications. The specifics of the API will be discussed in the next section.

The architecture described in Figure 22 does not have to be manually created by each user wanting to run this application. The architecture was implemented with Docker, which is a lightweight container system that is platform independent. That means one user can run this Malgazer architecture on MacOS, while another can run it on Linux or Microsoft Windows, without changing any of Malgazer's source code on either installation. The creation of the Docker containers is completely scripted, so this also makes it possible to easily push this architecture to the cloud since all software library dependencies were satisfied internally within each container.

The file defining the architecture in Figure 22 is "docker-compose.yml" in the Malgazer source code repository. This file specifies each node in the architecture, the configuration, the environment variables, and anything else a network requires. The Docker compose file is read when the commands "docker-compose build" and "docker-compose up" are executed. The first command builds the container images from the included "Dockerfile" file. The second command runs the container images and creates the architecture in Figure 22 on a virtual

Figure 22. The web application architecture

network. The Docker compose file then routes HTTPS traffic to the Nginx load balancer. The Nginx load balancer decides where the traffic will head next. If the URL begins with “api” after the host name, the traffic is routed to the API node. All other traffic is routed to the web application node.

Environment variables are the main method to configure the microservices in Figure 22. The Docker compose file will read the “.env” file in the local directory for the required environment variables. This file will be specific to each user, and it must be created. A template with the name “.env.template” is available in the repository so that users can copy it to “.env” and use it as a guide to get started. The content of the environment variable template file follows:

```
01.# DB settings...
02.POSTGRES_PASSWORD=malgazer
03.
04.# Set to 1 to use Malgazer in multi-user mode.
05.MALGAZER_MULTIUSER=0
06.
07.# Set to dev or prod
08.MALGAZER_RUN_ENV=dev
09.
10.# General settings...
11.MALGAZER_WEB_SECRET_KEY=malgazer
12.MALGAZER_WEB_SECURITY_PASSWORD_SALT=malgazer
13.MALGAZER_WEB_MAIL_USERNAME=YOURUSERNAME
14.MALGAZER_WEB_MAIL_PASSWORD=PASSWORD
15.MALGAZER_WEB_MAIL_SERVER=YOURSERVER
16.MALGAZER_WEB_MAIL_PORT=YOURPORTNUMBER
17.MALGAZER_WEB_MAIL_USE_SSL=1
18.MALGAZER_WEB_MAIL_USE_TLS=0
19.MALGAZER_WEB_HISTORY_LENGTH=100
20.MALGAZER_ALLOW_RESET=1
21.
22.# Production values...
23.MALGAZER_API_WORKERS=2
24.MALGAZER_API_THREADS=10
25.MALGAZER_WEB_WORKERS=2
26.MALGAZER_WEB_THREADS=10
27.
28.# For docker stack...
29.MALGAZER_API_REPLICAS=2
30.MALGAZER_WEB_REPLICAS=2
```

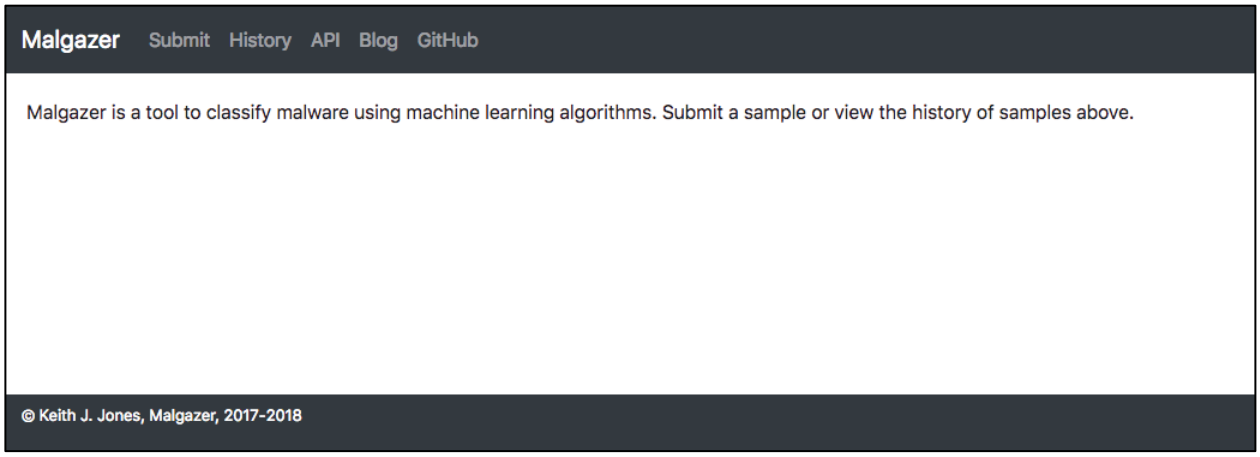


Figure 23. The Malgazer web application main screen

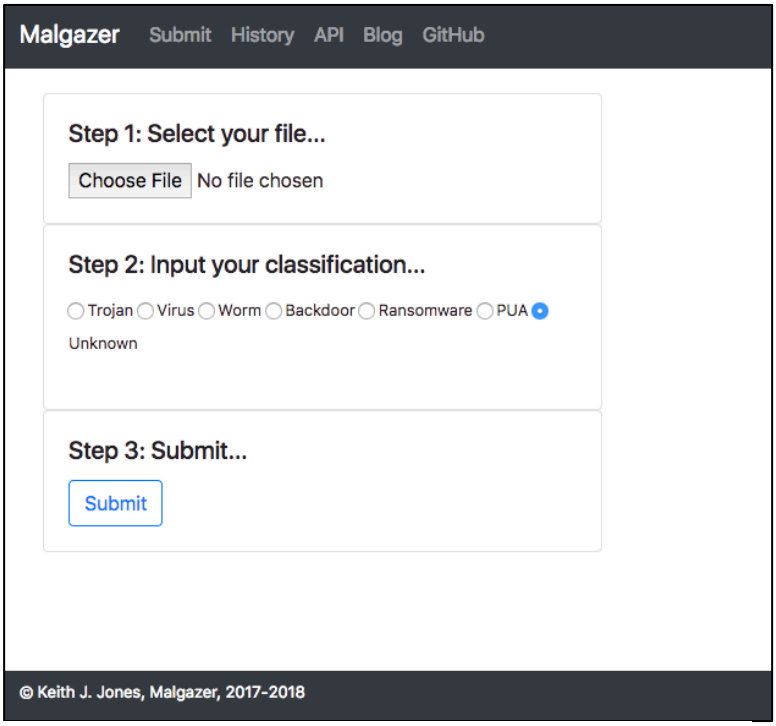


Figure 24. The Malgazer submission form

Line 01 is the password for the Postgres database. Users should change this variable to a secure password. If users wish to run the Malgazer web application in multi-user mode, the variable on line 05 must be changed from "0" to "1". This will enable more functionality such as account management, rate limiting, and logins. The multi-user mode will not be discussed in this dissertation, but it is very similar to the single-user mode.

Line 08 configures the application as either development (more verbose) or production (less verbose). The variables in lines 11-12 are security settings, and the user should change the

values to a secure password. Lines 13-18 configure email support, which is required if the Malgazer web application is started in multi-user mode. Line 19 configures how many lines will be in the classification history window. Line 20, if set to "1", will allow users to reset the database remotely. This should be disabled (a "0") in production. Lines 23-26 configure the number of workers and threads for the application and can be left to the default values in most cases. Lastly, lines 29-30 configure the number of replicas, if the user wishes to use Docker stack to deploy this web application to a swarm.

Once the environment variables have been set, the application can be started with the "docker-compose up" command. Navigating a web browser to https://localhost displays the main screen, as shown in Figure 23. Note that the screen captures do not show the multi-user portions of this application, for simplicity. Next, a user can submit a sample by clicking "Submit" in the menu, which provides an upload form in Figure 24. Submitted samples are saved as files named as their SHA256 value in the "samples" directory of the source code repository.

After selecting and uploading a number of samples to the Malgazer web application, the user can view the scan history. The scan history includes the predicted classification by the Malgazer classifier loaded from the pickled file named "ml.dill" in the repository's "classifier" directory. This "ml.dill" file was created from the Malgazer source code during the final training phase for each type of classifier studied in this research. These files are available via a link in

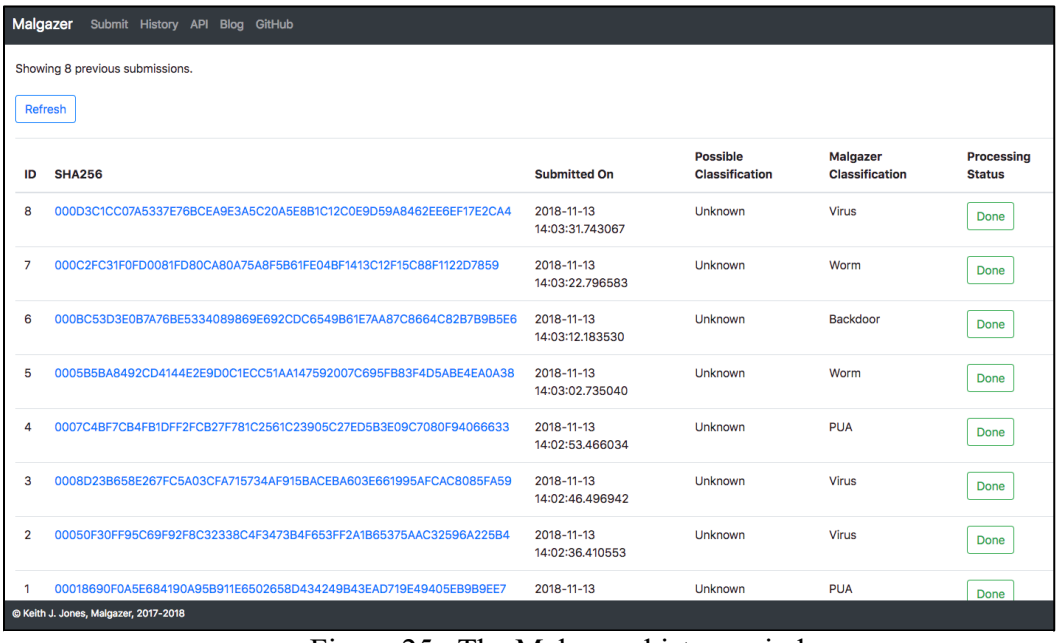


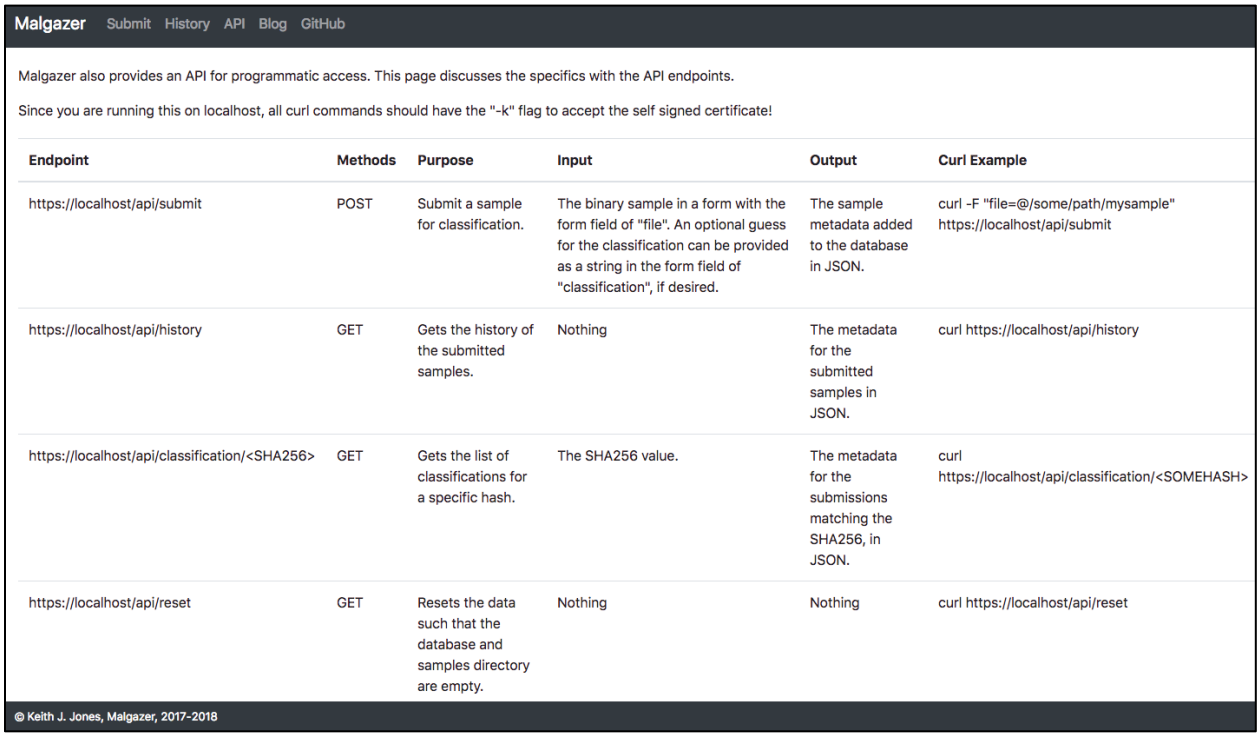
Figure 25. The Malgazer history window

the Malgazer source code repository. Simply replacing this file with other classifiers trained with the Malgazer source code will allow a user to easily interact with all of the classifiers through a web browser. Figure 25 presents the history for several example malware samples from the final data set, complete with their predicted classifications. "Possible Classifications" is a value that can be used to store the human prediction of the same sample. This value can be set when the sample is submitted, or simply ignored because it has no bearing on the predicted classification.

The web application is a Python Flask application, and it was designed to be a thin wrapper over the Malgazer API. This allows users to not only use the convenient web application, but users are also able to use the backend API programmatically if they wish to submit larger collections of samples very quickly. All of the functionality found in the web application is also available through the API. The API is documented in the web application by clicking "API" at the top of the menu, leading to the window presented in Figure 26. The next

Figure 26. The Malgazer API documentation

section will briefly discuss the Malgazer API.



Malgazer Submit History API Blog GitHub

Malgazer also provides an API for programmatic access. This page discusses the specifics with the API endpoints.

Since you are running this on localhost, all curl commands should have the "-k" flag to accept the self signed certificate!

Endpoint	Methods	Purpose	Input	Output	Curl Example
https://localhost/api/submit	POST	Submit a sample for classification.	The binary sample in a form with the form field of "file". An optional guess for the classification can be provided as a string in the form field of "classification", if desired.	The sample metadata added to the database in JSON.	curl -F "file=@/some/path/mysample" https://localhost/api/submit
https://localhost/api/history	GET	Gets the history of the submitted samples.	Nothing	The metadata for the submitted samples in JSON.	curl https://localhost/api/history
https://localhost/api/classification/<SHA256>	GET	Gets the list of classifications for a specific hash.	The SHA256 value.	The metadata for the submissions matching the SHA256, in JSON.	curl https://localhost/api/classification/<SOMEHASH>
https://localhost/api/reset	GET	Resets the data such that the database and samples directory are empty.	Nothing	Nothing	curl https://localhost/api/reset

© Keith J. Jones, Malgazer, 2017-2018

The Malgazer API

The Malgazer API is the engine behind the classification process for the entire web application. There are four main API routes: submit a sample, see the history of classifications, get the classifications for a specific SHA256 hash, and reset the database. Most of the classification logic is in the submission route.

The submission route contains logic to read the file from the upload, load the “ml.dill” pickled object into memory, and use that object to generate the features and classifications for the uploaded malware samples. The submission route then saves the results to the database so it is available in the “history” API route. The submission route uses a fixed sized process pool to calculate the features and classifications in parallel, decreasing the overall time required to classify multiple malware samples. Using a pool also prevents resource depletion and allows tuning for the number of simultaneous jobs executed per Docker container.

The API returns JSON so it can be easily parsed using nearly any popular programming language. The following two lines demonstrate the API usage. Using the command line web browser named “curl” in line 01, the JSON classification history in line 02 was produced. Line 02 was the JSON output from the API. The API’s JSON output matched the HTML version in

Figure 25. Line 02 could be loaded with “json.loads” function inside the Python interpreter, if one wished, to access attributes about each scan as a dictionary.

```
01.$ curl -k https://localhost/api/history
02.[{"id": 8, "sha256":
   "000D3C1CC07A5337E76BCEA9E3A5C20A5E8B1C12C0E9D59A8462EE6EF17
   E2CA4", "time": "2018-11-13 14:03:31.743067",
   "classification": "Virus", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 7, "sha256":
   "000C2FC31F0FD0081FD80CA80A75A8F5B61FE04BF1413C12F15C88F1122
   D7859", "time": "2018-11-13 14:03:22.796583",
   "classification": "Worm", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 6, "sha256":
   "000BC53D3E0B7A76BE5334089869E692CDC6549B61E7AA87C8664C82B7B
   9B5E6", "time": "2018-11-13 14:03:12.183530",
   "classification": "Backdoor", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 5, "sha256":
   "0005B5BA8492CD4144E2E9D0C1ECC51AA147592007C695FB83F4D5ABE4E
   A0A38", "time": "2018-11-13 14:03:02.735040",
   "classification": "Worm", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 4, "sha256":
   "0007C4BF7CB4FB1DFF2FCB27F781C2561C23905C27ED5B3E09C7080F940
   66633", "time": "2018-11-13 14:02:53.466034",
   "classification": "PUA", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 3, "sha256":
   "0008D23B658E267FC5A03CFA715734AF915BACEBA603E661995AFCAC808
   5FA59", "time": "2018-11-13 14:02:46.496942",
   "classification": "Virus", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 2, "sha256":
   "00050F30FF95C69F92F8C32338C4F3473B4F653FF2A1B65375AAC32596A
   225B4", "time": "2018-11-13 14:02:36.410553",
   "classification": "Virus", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"},
   {"id": 1, "sha256":
   "00018690F0A5E684190A95B911E6502658D434249B43EAD719E49405EB9
   B9EE7", "time": "2018-11-13 14:02:28.892873",
   "classification": "PUA", "possible_classification":
   "Unknown", "ip_address": "172.30.0.5", "status": "Done"}]
```

One can submit a new malware sample with the following curl command (note that the line number is not part of the command):

```
01.curl -F "file=@/some/path/mysample"
   https://localhost/api/submit
```

The command above uploads the "mysample" binary to the API so that it can be classified. Once the results have finished processing in the API process pool, calling the "history" route will display the predicted classification information. One can also pull classifications for a specific SHA256 using the "classification" API route.

In multi-user mode, all of the API routes can be protected with rate-limiting, so that single users cannot deplete all of the computing resources. The rate limiting can be applied to each user in this manner. For users that wish to disable rate limiting, it can be turned off completely in the source code.

Summary

This chapter briefly demonstrated the web application created to facilitate usage of the Malgazer classifiers easily from a web browser or through an API. This chapter highlighted the architecture, how this application works, and the commands used to create it. Example output from the web application and the API were also included in this chapter. The Malgazer web application was the final design science artifact created by this research. The next and final chapter will present the conclusions for the entire dissertation.

CHAPTER 9

CONCLUSIONS

Introduction

This chapter will explore the conclusions drawn from this research project. While previous chapters concentrated on the details of each experiment, this chapter will present the conclusions drawn from this research project as a whole. First, a number of conclusions can be drawn based upon the design science artifacts initially presented in the first chapter. This chapter will summarize the research contributions and then state the conclusions drawn for each contribution. Then, this chapter will present some of the benefits and disadvantages observed from using the various techniques presented throughout this dissertation. This section may be useful to readers that are undecided as to how they will develop their classifier because it will provide insights beyond the raw data. This chapter will conclude with several possible exciting future research directions discovered during this research.

The Research Contributions

There were several research objectives presented in the first chapter that created design science artifacts as contributions to science. Recall that this research began with the assertion that a generalized malware classification model could be used to explain prior literature for malware classification. This model also allowed for the comparison of prior literature to the research developed in this dissertation. Recall that as part of this dissertation's research question, running window entropy was chosen as the input data to the classifier. However, it was discovered that running window entropy data was very slow to compute at scale. Therefore, this dissertation optimized the running window entropy algorithm so that it could be calculated faster on larger data sets. After the running window entropy data was computed, the collection and normalization of the six functional malware classifications based upon VirusTotal data was completed.

After obtaining the training data, the training phase was executed at scale so that one individual was able to complete over two hundred complex training experiments in parallel in the cloud. In order to train and evaluate the classifiers at scale, this research used Terraform to instantiate cloud computing resources as needed. Once the classifiers were trained and the accuracies measured, it was determined that the most accurate classifier was the Adaboost/RF algorithm with running window entropy data as input. This classifier had an accuracy of approximately 95%, which was approximately 0.76% greater than the measured accuracy of the GIST model using the same Malgazer data sets. Using the accuracies measured from the GIST and Malgazer classifiers, in addition to the accuracies reported in prior literature, a high-level comparison of Malgazer methods to prior literature was presented.

The final design science artifact presented from this research was the web application. The web application allowed users without Python programming familiarity to utilize the classifiers built from this research. More importantly, the web application also encapsulated all of the dependencies within Docker so that it could run on nearly any operating system. Each of these contributions and their conclusions will be discussed in their own subsections that follow.

The Generalized Classification Model

The generalized machine learning based malware classification model was developed during the literature review phase. As the literature review began, it quickly became apparent that each article was written independently from others, and comparisons between methods proposed in different articles were difficult, or in some cases impossible, to discern. In order to compare two methods, a common framework was needed. This framework was expressed in this dissertation as a generalized malware classification model. The purpose of this generalized model was to aid in the categorization of numerous machine learning malware papers relevant to this research topic. Based upon over one hundred articles considered, the model presented in the first chapter was developed and refined.

The first conclusion from this generalized model was that it successfully allowed for the comparison of prior methods not only to each other, but also to the methods presented in this dissertation. Without this model, it would have been difficult to visualize the differences between methods. With this model, it was possible to categorize and present the prior literature. In some cases, the comparisons were as trivial as comparing the value of two cells in a table, but

in other cases authors did not provide enough information for comparison. This design science research artifact served its purposes for this dissertation, but it also gives future researchers a powerful tool to compare their research to prior literature, as was done here.

The biggest disappointment from using the generalized model was discovering a great number of prior papers omitting critical information that would have been useful for replication, verification, and comparative purposes. In cases where there was enough information to adequately replicate a prior method, the malware training data set was not available. Therefore, replication, verification, and comparisons were cursory, at best, except for the GIST method.

Calculating the accuracies for the GIST malware classifier on the same Malgazer data set allowed for direct comparisons between the GIST and Malgazer methodologies. Indirectly, it also compared the GIST and Malgazer training and testing data sets. This is because the training and testing data sets directly influence the accuracy scores. As discussed previously, a poorly selected training or testing data set could cause the classifier to learn features of malware that may not be representative of the real world. The data sets could also artificially inflate reported accuracies if the testing and training data sets are too much alike. In other cases, a training data set could reduce the accuracy if it did not contain enough features useful for classification. Using the same data sets to compare the GIST and Malgazer methodologies removed a variable in the comparison.

Comparing the GIST and Malgazer classifiers on the same data sets also allowed for cursory theoretical comparisons to other prior literature. For example, the accuracies reported in this dissertation were lower for GIST than the corresponding accuracies reported in their original paper of approximately 98%. As discussed in the analysis chapter 7, if one were to assume that the data set in this dissertation was not optimal, it is possible that the GIST and Malgazer classifiers could have been more accurate if they were built from a different malware data set. The results of the GIST papers support that assertion, as the authors had a data set that led to an accuracy of approximately 98%. Comparing GIST and Malgazer using the same malware data set discovered that the Malgazer classifier could outperform the GIST methodology if Adaboot and random forests were used to classify the running window entropy data from the samples. The accuracy improvement was approximately 0.75%, which is not large, but it is still an improvement. At a minimum, it is fair to concluded that the Malgazer classifier is at least as accurate as the GIST classifier.

Using the comparison between GIST and Malgazer, a theoretical comparison to other prior literature based solely on the reported accuracies was made. Most prior papers presented accuracies of approximately 98%, which was also the accuracy reported by the GIST authors. Since the Malgazer classifier outperformed the GIST classifier, and the GIST authors reported an accuracy of approximately 98%, it is possible that Malgazer's true accuracy could be closer to 98% if the training and testing data sets were selected to optimize this accuracy. However, the data for this dissertation was selected from a stream of real-world current threats, and that stream could have either matched the general trend in threats in 2018, or it could have been an outlier and biased the classifier's training. No matter the case, it is important to note that the extra 3% accuracy could be gained using a different malware data set since the GIST and Malgazer accuracies were very close for this dissertation's data set.

The selection of the functional classification also directly influenced the accuracies. As discussed in the analysis chapter, the "Trojan" functional classification produced the worst accuracy every time. Previously, this dissertation explored some of the reasons that a "Trojan" classification could have been problematic. Therefore, higher accuracy may be in reach by removing the "Trojan" classification and associated samples.

The generalized malware classification model design science artifact satisfied this research objective.

Running Window Entropy Optimization

The running window entropy optimization allowed for more data to be computed in less time. Therefore, this optimization allowed for larger training and testing data sets. The conclusions from this research objective are relatively straight forward. The optimized running window entropy algorithm required 2% or less of the time required for the original algorithm. Therefore, fifty times the data could be computed by the optimized algorithm over the original algorithm. Since more data could be computed in the same amount of time, the training and testing data sets were larger than they may have been without the optimized algorithm. Larger training data sets typically lead to more accurate classifiers.

The design science method artifact of an optimized running window entropy algorithm satisfied this research objective.

Collection And Normalization of Functional Classification Information

Another pivotal decision during this research project that affected the overall accuracies of classifiers was the selection of the six functional malware classifications. Six functional classifications were chosen based upon the VirusTotal detections of the exploratory data sets. Six categories were used because it focused the research question, but there are many more than six functional categories for malware. Using a small number of categories with the selection of categories directly impacted the classification accuracies of any such classifier. A number of prior methods, such as GIST, used many more categories and resulted in higher accuracies.

As discussed previously, the "Trojan" classification consistently produced the worst accuracy out of all the categories. Removing this classification would most likely increase the classifier's accuracy. Many possibilities were explored as to why the "Trojan" classification may be problematic, and users interested in the details should refer back to chapter 7.

The method of collecting and normalizing functional classification information is a design science method artifact. This design science method artifact satisfied this research objective.

Scaling The Computations

The process of scaling the computations in the cloud made it possible to collect data from over two hundred training sessions. The two hundred training sessions only required the assistance of one individual, so the methodologies presented throughout this dissertation can easily be replicated by nearly anyone if they explore the Malgazer repositories. Using cloud resources to compute the experiments throughout this dissertation cost less than $1,000 USD total[5], so the costs were not prohibitive for most organizations. Using cloud resources allowed for multiple simultaneous experiments to be calculated in parallel, which would have taken much longer to compute on a single laptop and cost much more for hardware.

Furthermore, cloud resources provided hardware that may not be on the average user's computer. For example, the neural networks computed quickly because a cloud resource with a GPU was selected for the experiments. The GPU reduced the computation time greatly, but in

[5] This was a best guess estimate of the total cost from this author's Azure account, this author's AWS account, and the committee chair's AWS account. Several accounts were used for the training, but the overall costs were not over $1,000 USD.

2017-2018 the demand for GPUs greatly outpaced the available supply. The cost of a single GPU during these years were in the thousands of dollars because of the cryptocurrency boom. The average cost of a GPU during the boom was many times more expensive than an average GPU before the boom. Amazon AWS provided a "spot instance" with a GPU for a much lower cost than normal, and that instance was used for efficient neural network training. Cloud resources made it possible to efficiently train classifiers with a GPU without the large upfront costs of buying the hardware outright.

For the traditional machine learning algorithms, it was important to use computers with large memory and numerous CPU cores. Microsoft Azure and Amazon AWS resources provided computers with these specifications at a lower cost than purchasing the hardware outright. Depending on the experiments ran at the time, cloud computers with eight or sixteen CPUs were selected, with matching large memory allocations.

While it might not seem important that a prior science artifact was used to scale the computations, it was very important to the success of this research project. Given that most dissertations only last approximately one year from proposal to defense, and they are written by one author, efficient use of time was very important. The method of scaling these computations so that one individual could complete them within a year was directly responsible for the success in collecting the large amounts of data for this dissertation in a timely manner. This research objective was satisfied by this design science instantiation artifact.

The Most Accurate Classifier

The most accurate classifier based on the Malgazer training and testing data sets came from the Adaboost and random forest algorithms using the running window entropy data as input features. The accuracy for this classifier was approximately 95%, which means that on average out of 100 classifications, approximately five of the classifications will be errors. The most accurate classifier was a slight improvement over the GIST classifier, making it at least as effective as the GIST classifier. This classifier satisfies one of the design science research objectives described in the first chapter.

Other conclusions drawn previously in this chapter about this classifier will not be restated here in order to save space, but the conclusions still apply nonetheless.

The Web Application

The last design science artifact was the web application. The web application used several systems to provide a user friendly and programmatic access to the classifiers trained throughout this research. The web application was designed so that it runs in Docker, which means all of the dependencies are encapsulated in the application and it can be run on nearly any platform, such as Microsoft Windows, Apple MacOS, and Linux. Once built, the web application can run in single user or multiple user mode. The application can also be easily pushed to the cloud, since Docker also runs in the cloud for Azure and AWS.

The web application design science artifact satisfied the research objective described in the first chapter.

Recommendations

Until now, this dissertation concentrated on reporting the numbers, facts, and results. However, some users may be looking for recommendations for selecting the right malware classifier for a specific application. This section will provide some of this author's recommendations based on the benefits and disadvantages learned during the duration of this research project.

First, it was quickly obvious that running window entropy required more computation time than GIST. Since GIST and Malgazer classifiers were nearly the same for accuracy, the extra computation time may be onerous for some organizations and not worth the tradeoff for higher accuracy. Other organizations may already have been using running entropy data during their malware analysis, so the addition of the Malgazer classifier on top of that data would be more efficient than adding a GIST classifier. In this case, adding a GIST classifier would require extra computations and possible storage of this GIST information as well. Therefore, time constraints will factor into which classifier a user should choose, and what type of environment is already available to the user (i.e. is RWE data already calculated, or not?).

If a user is only interested in the highest accuracy possible, the Malgazer classifier using the Adaboost and random forest algorithms on the running window entropy data was the most accurate classifier created by this research. This classifier was approximately 1% more accurate

than the GIST method on the same data sets. The tradeoff for more accuracy in this case was the longer computation time for the running window entropy data calculations.

If a user is not interested in developing their own source code to support a malware classifier, the user's choices are extremely limited. Very few, if any, prior literature reviewed in this dissertation provided working source code without first contacting the authors. On the other hand, everything in this dissertation has been open sourced and is available to the public. Any user comfortable with Python, the web, or APIs could make use of the source code, data, and any of the other design science artifacts presented throughout this dissertation. Improvements will be welcomed to the Malgazer repositories.

While the GIST data size was fixed at 320 features, the running window entropy data size was dependent on the running window size and the malware's size. For supervised machine learning, the number of inputs must be constant between samples, so the running window entropy data had to be resampled to a fixed length. This dissertation explored several resampled lengths, from 256 bytes up to 4,096 bytes. While this research did not exhaustively search the running window entropy window size and resampled length possibilities, it did not appear that the accuracy would change greatly when using different lengths. Therefore, a negligible difference in classification accuracy[6] for using a smaller running window size of 256 bytes and feature vector of 512 bytes was a good tradeoff for the faster computation time. A running window size of 256 bytes and an RWE vector size of 512 bytes are the values this author recommends using as a starting point.

Future Research Opportunities

Several ideas for future research opportunities are presented in this section. First, exploring methods to curate an optimal training and testing data sets could improve an already high classification accuracy score. As discussed previously, the classifier's accuracy is dependent on the input training and testing data sets, and the methodology to curate such a set for this classifier could be studied. Another future research opportunity is study into the optimal functional classification categories. In this dissertation, the six functional classifications were drawn from visual observation of VirusTotal data. Perhaps research into this topic could also

[6] The differences in classification accuracy were approximately 1% or less.

improve the Malgazer classifier's accuracy. Additional research could explore the differences in accuracies between classification strategies. For example, the accuracies for functional classifications could be compared to family classifications.

While this dissertation considered a large number of prior works, there are prior works that this dissertation did not cover. In addition, there will also be many future works that could be classified using the generalized classification model. Unless there is wide adoption of a model such as this to describe methodologies, publications classifying the current research such as was done in the literature review chapter could be beneficial to others.

Regarding prior literature, there is also a need to verify methods for the purposes of comparison. This dissertation explored one such method in GIST, but there are many more methods that should be replicated, verified, and compared to current literature so that scientists can truly measure and understand improvements to malware classifiers.

Although discussed briefly in the last section, future research could concentrate on the relationships between RWE window size and the resampled RWE length with respect to accuracy. Although this research did not discover any correlation to accuracy, the anaylsis was cursory and further study is possible in that area.

With respect to the Malgazer web application, another future research opportunity could be to build an online malware classifier that learns as users submit samples. Online classifiers learn with each new sample instead of every sample in the training set up front, as was discussed throughout this dissertation.

Another future research opportunity could explore the relationships between automated malware classification and human based malware classification. There are many factors that could be studied, such as the differences in accuracies between a classifier and a human, and the time savings versus the accuracy gain an organization could use to their benefit.

Summary

This chapter presented the conclusions drawn from the research project described in this dissertation. The research objectives from the first chapter were revisited and each design science artifact was discussed with respect to how it satisfied the objectives. Then, this chapter presented several recommendations based upon this author's experience working on this

dissertation topic between 2016 and 2019. Lastly, this chapter presented future research opportunities that could be based on the work described in this dissertation.

DEFINITIONS

Adaboost	The Adaboost machine learning algorithm.
AI	Artificial intelligence.
ANN	Artificial neural networks.
CNN	Convolutional neural networks.
DT	Decision tree machine learning algorithm.
GIST	Method #14 in the literature review chapter.
KNN	K-Nearest neighbors machine learning algorithm.
Malgazer	A name given to the classifiers developed in this research.
ML	Machine learning.
NB	Naïve Bayes machine learning algorithm.
NC	Nearest Centroid machine learning algorithm.
OVR	One v Rest machine learning algorithm.
RF	Random forest machine learning algorithm.
RWE	Running window entropy.
SVM	Support vector machines machine learning algorithm.

REFERENCES

Annachhatre, C., Austin, T. H., & Stamp, M. (2013). *Hidden Markov models for malware classification*. San Jose State University. https://doi.org/10.1007/s11416-014-0215-x

Arnould, E. J., Hevner, A. R., March, S. T., & Park, J. (2004). Design Science in Information Systems Research. *MIS Quarterly*. https://doi.org/10.2307/25148869

Athiwaratkun, B., & Stokes, J. W. (2017). Malware Classification with LSTM and GRU Language Models and a Character-Level CNN. *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference On*, 2482–2486.

Azmoodeh, A., Dehghantanha, A., & Choo, K.-K. R. (2018). Robust Malware Detection for Internet Of (Battlefield) Things Devices Using Deep Eigenspace Learning. *IEEE Transactions on Sustainable Computing*, 1–1. https://doi.org/10.1109/TSUSC.2018.2809665

Baysa, D. (2013). *Structural Entropy and Metamorphic Malware*. San Jose State University.

Burnap, P., French, R., Turner, F., & Jones, K. (2018). Malware classification using self organising feature maps and machine activity data. *Computers and Security*, *73*, 399–410. https://doi.org/10.1016/j.cose.2017.11.016

Cakir, B., & Dogdu, E. (2018). Malware Classification Using Deep Learning Methods. *Proceedings of the ACMSE 2018 Conference*, 1–7.

Choi, S., Jang, S., Kim, Y., & Kim, J. (2017). Malware detection using malware image and deep learning. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)* (pp. 1193–1195). https://doi.org/10.1109/ICTC.2017.8190895

Cylance Inc. (2016). CylancePROTECT® is the First Signature-less Next Generation Antivirus to be Certified by AV-TEST. Retrieved February 10, 2018, from https://threatvector.cylance.com/en_us/home/cylanceprotect-is-the-first-signature-less-next-generation-antivirus-to-be-certified-by-av-test.html

Dahl, G. E., Stokes, J. W., Deng, L., & Yu, D. (2013). Large-scale malware classification using random projections and neural networks. *2013 IEEE International Conference on Acoustics,*

Speech and Signal Processing, 3422–3426. https://doi.org/10.1109/ICASSP.2013.6638293

Docker Inc. (n.d.). Docker. Retrieved April 8, 2018, from https://www.docker.com/

Eremenko, K., & de Ponteves, H. (2017). Deep Learning A-Z. Retrieved February 17, 2018, from https://www.udemy.com/deeplearning/

Gharacheh, M., Derhami, V., Hashemi, S., & Fard, S. M. H. (2016). Detection of Metamorphic Malware based on HMM: A Hierarchical Approach. *International Journal of Intelligent Systems and Applications*, *8*(4), 18–25. https://doi.org/10.5815/ijisa.2016.04.02

Gibert, D., & Bejar, J. (2016). *Convolutional Neural Networks for Malware Classification*. Escola Politècnica de Catalunya (UPC) - BarcelonaTech. Retrieved from http://www.covert.io/research-papers/deep-learning-security/Convolutional Neural Networks for Malware Classification.pdf

Gu, B., Fang, Y., Jia, P., Liu, L., Zhang, L., & Wang, M. (2015). A New Static Detection Method of Malicious Document Based on Wavelet Package Analysis. In *2015 International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)* (pp. 333–336). IEEE. https://doi.org/10.1109/IIH-MSP.2015.72

HaddadPajouh, H., Dehghantanha, A., Khayami, R., & Choo, K. R. (2018). A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting. *Future Generation Computer Systems*. https://doi.org/10.1016/j.future.2018.03.007

Han, R. (2017). *Data-Driven Malware Detection Based on Dynamic Behavioral Features*. University of Miami. Retrieved from http://scholarlyrepository.miami.edu/oa_dissertations%0Ahttp://scholarlyrepository.miami.edu/oa_dissertations/1806

Hanley, J. A., & McNeil, B. J. (1982). The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve. *Radiology*, *143*(1), 29–36. https://doi.org/10.1148/radiology.143.1.7063747

Hassen, M. S. (2018). *Machine Learning for Classifying Malware in Closed-set and Open-set Scenarios*. Florida Institute of Technology. Retrieved from https://cs.fit.edu/~pkc/theses/hassen18.pdf

Huang, T. H., & Kao, H. (2017). R2-D2: ColoR-inspired Convolutional NeuRal Network

(CNN)-based AndroiD Malware Detections. Retrieved from http://arxiv.org/abs/1705.04448

Huang, W., & Stokes, J. W. (2016). MtNet: A Multi-Task Neural Network for Dynamic Malware Classification. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 9721, pp. 399–418). https://doi.org/10.1007/978-3-319-40667-1_20

Jones, K. (2017a). Algorithms #1 Python Source Code. Retrieved June 18, 2017, from https://github.com/keithjjones/csc705-alg1

Jones, K. (2017b). Malgazer. Retrieved from https://github.com/keithjjones/malgazer

Jones, K. (2018). Malgazer Terraform Code. Retrieved November 1, 2018, from https://github.com/keithjjones/malgazer-terraform

Jones, K., & Wang, Y. (2018). An Optimized Running Window Entropy Algorithm. In *2018 National Cyber Summit (NCS)*. Huntsville, AL: IEEE. https://doi.org/10.1109/NCS.2018.00016

Kalash, M., Rochan, M., Mohammed, N., Bruce, N. D. B., Wang, Y., & Iqbal, F. (2018). Malware Classification with Deep Convolutional Neural Networks. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 1–5. https://doi.org/10.1109/NTMS.2018.8328749

Keras. (n.d.). Retrieved February 17, 2018, from http://keras.io

Kilgallon, S., De La Rosa, L., Cavazos, J., Rosa, L. D. La, & Cavazos, J. (2017). Improving the effectiveness and efficiency of dynamic malware analysis with machine learning. *2017 Resilience Week (RWS)*, 30–36. https://doi.org/10.1109/RWEEK.2017.8088644

Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., & Eckert, C. (2017). Empowering convolutional networks for malware classification and analysis. *Proceedings of the International Joint Conference on Neural Networks*, *2017–May*, 3838–3845. https://doi.org/10.1109/IJCNN.2017.7966340

Kolosnjaji, B., Zarras, A., Webster, G., & Eckert, C. (2016). Deep Learning for Classification of Malware System Call Sequences. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol.

9992 LNAI, pp. 137–149). https://doi.org/10.1007/978-3-319-50127-7_11

Kolter, J. Z., & Maloof, M. A. (2004). Learning to detect malicious executables in the wild. *Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '04*, 470. https://doi.org/10.1145/1014052.1014105

Kolter, J. Z., & Maloof, M. A. (2006). Learning to detect malicious executables. In *Machine Learning and Data Mining for Computer Security* (pp. 47–63). Springer.

Krcal, M., Svec, O., & Balek, M. (2018). Deep Convolutional Malware Classifiers Can Learn From Raw Executables And Labels Only. *ICLR 2018*. Retrieved from https://openreview.net/pdf?id=HkHrmM1PM

Laks. (2013). Finding Visually Similar Malware among Millions of Malware. Retrieved June 30, 2018, from http://sarvamblog.blogspot.com/2013/10/finding-visually-similar-malware-among.html

Laks. (2014). Supervised Classification with k-fold Cross Validation on a Multi Family Malware Dataset. Retrieved June 30, 2018, from http://sarvamblog.blogspot.com/2014/08/supervised-classification-with-k-fold.html

Lall, A., Sekar, V., Ogihara, M., Xu, J., & Zhang, H. (2006). Data Streaming Algorithms for Estimating Entropy of Network Traffic. *SIGMETRICS Perform. Eval. Rev.*, *34*(1), 145–156. https://doi.org/10.1145/1140103.1140295

Le, Q., Boydell, O., Mac, B., & Scanlon, M. (2018). Deep Learning at the Shallow End : Malware Classification for Non-Domain Experts. *Digital Investigation*.

Lui, L., & Wang, B. (2017). Automatic Malware Detection Using Deep Learning Based on Static Analysis. In *Data Science* (pp. 500–507). International Conference of Pioneering Computer Scientists, Engineers and Educators. https://doi.org/10.1145/3015456

Lyda, R., & Hamrock, J. (2007). Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security & Privacy*. https://doi.org/10.1109/MSP.2007.48

McLaughlin, N., Doupé, A., Joon Ahn, G., Martinez del Rincon, J., Kang, B., Yerima, S., … Zhao, Z. (2017). Deep Android Malware Detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY '17* (pp. 301–308). https://doi.org/10.1145/3029806.3029823

McMillan, C., & Garman, J. (2007). System and method for determining data entropy to identify malware. United States. Retrieved from https://patents.google.com/patent/US8069484

Meng, X., Shan, Z., Liu, F., Zhao, B., Han, J., Wang, H., & Wang, J. (2017). MCSMGS: Malware Classification Model Based on Deep Learning. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* (pp. 272–275). https://doi.org/10.1109/CyberC.2017.21

Mosli, R., Li, R., Yuan, B., & Pan, Y. (2017). A Behavior-Based Approach for Malware Detection. In *13th IFIPWG 11.9 International Conference*. Orlando, Florida, USA. https://doi.org/10.1007/978-3-642-24212-0

Narra, U. (2015). *Clustering versus SVM for Malware Detection*. San Jose State University.

Narra, U., Troia, F. Di, Corrado, V. A., Austin, T. H., & Stamp, M. (2016). Clustering versus SVM for malware detection. *Journal of Computer Virology and Hacking Techniques*, *12*(4), 213–224. https://doi.org/10.1007/s11416-015-0253-z

Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware images: visualization and automatic classification. *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, 4. https://doi.org/10.1145/2016904.2016908

Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware images: visualization and automatic classification. *Proceedings of the 8th International Symposium on Visualization for Cyber Security*. Pittsburgh, Pennsylvania, USA: ACM. https://doi.org/10.1145/2016904.2016908

Nataraj, L., Karthikeyan, S., & Manjunath, B. S. (2015). SATTVA: SpArsiTy Inspired classificaTion of Malware VAriants. *Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security*, 135–140. https://doi.org/10.1145/2756601.2756616

Nath, H. V, & Mehtre, B. M. (2015). Ensemble learning for detection of malicious content embedded in PDF documents. In *Signal Processing, Informatics, Communication and Energy Systems (SPICES), 2015 IEEE International Conference on* (pp. 1–5). https://doi.org/10.1109/SPICES.2015.7091371

Ni, S., Qian, Q., & Zhang, R. (2018). Malware Identification Using Visualization Images and Deep Learning. *Computers & Security*. https://doi.org/10.1016/j.cose.2018.04.005

Pai, S., Troia, F. Di, Visaggio, C. A., Austin, T. H., & Stamp, M. (2017). Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques*, *13*(2), 95–107. https://doi.org/10.1007/s11416-016-0265-3

Panda Security. (2017). *2017 Cybersecurity Predictions*. Retrieved from http://www.pandasecurity.com/mediacenter/src/uploads/2017/02/Pandalabs-2017-Predictions-en.pdf

Paul, C. B. (2017). *Entropy-based file type identification and partitioning*. Naval Postgraduate School. Retrieved from http://hdl.handle.net/10945/55513

Pektaş, A., & Acarman, T. (2018). Malware classification based on API calls and behaviour analysis. *IET Information Security*, *12*(2), 107–117. https://doi.org/10.1049/iet-ifs.2017.0430

Perdisci, R., Lanzi, A., & Lee, W. (2008a). Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, *29*(14), 1941–1946. https://doi.org/10.1016/j.patrec.2008.06.016

Perdisci, R., Lanzi, A., & Lee, W. (2008b). McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 301–310. https://doi.org/10.1109/ACSAC.2008.22

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., & Nicholas, C. (2017). Malware Detection by Eating a Whole EXE. Retrieved from http://arxiv.org/abs/1710.09435

Rezende, E., Ruppert, G., Carvalho, T., Ramos, F., & Geus, P. De. (2017). Malicious Software Classification using Transfer Learning of ResNet-50 Deep Neural Network. *Journal of Computer Virology and Hacking Techniques 1*, *12*.

Rhode, M., Burnap, P., & Jones, K. (2017). Early Stage Malware Prediction Using Recurrent Neural Networks, (December), 1–28. Retrieved from http://arxiv.org/abs/1708.03513

Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic Analysis of Malware Behavior Using Machine Learning. *Journal of Computer Security*, *19*(4), 639–668. https://doi.org/10.3233/JCS-2010-0410

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., & Ahmadi, M. (2018). Microsoft Malware

Classification Challenge, 1–7. https://doi.org/10.1145/2857705.2857713

Saleh, M., Li, T., & Xu, S. (2018). Multi-context features for detecting malicious programs. *Journal of Computer Virology and Hacking Techniques*, *14*(2), 181–193. https://doi.org/10.1007/s11416-017-0304-8

Santos, I., Brezo, F., Ugarte-Pedrero, X., & Bringas, P. G. (2013). Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, *231*, 64–82. https://doi.org/10.1016/j.ins.2011.08.020

Saxe, J., & Berlin, K. (2015). Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features. *CoRR Abs/1508.03096*. https://doi.org/10.1109/MALWARE.2015.7413680

Schultz, M. G., Eskin, E., Zadok, E., & Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy - S&P* (pp. 38–49). https://doi.org/10.1109/SECPRI.2001.924286

scikit-learn. (n.d.). Retrieved February 17, 2018, from http://scikit-learn.org/

Scikit Learn AdaBoostClassifier. (n.d.). Retrieved December 8, 2018, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

Scikit Learn DecisionTreeClassifier. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Scikit Learn GaussianNB. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

Scikit Learn GridSearchCV. (n.d.). Retrieved July 25, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Scikit Learn KNeighborsClassifier. (n.d.). Retrieved December 8, 2018, from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

SciKit Learn LabelEncoder. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html

Scikit Learn NearestCentroid. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestCentroid.html

Scikit Learn OneHotEncoder. (n.d.). Retrieved December 8, 2018, from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

Scikit Learn OneVsRestClassifier. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html

Scikit Learn RandomForestClassifier. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Scikit Learn StandardScalar. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

Scikit Learn SVC. (n.d.). Retrieved December 8, 2018, from http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Shabtai, A., Moskovitch, R., Elovici, Y., & Glezer, C. (2009). Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*. https://doi.org/10.1016/j.istr.2009.03.003

Shafiq, M., Tabish, S., & Farooq, M. (2009). PE-probe: leveraging packer detection and structural information to detect malicious portable executables. *... of the Virus Bulletin Conference (VB)*, 1–10. Retrieved from http://nexginrc.org/nexginrcadmin/publicationsfiles/vb09-zubair.pdf

Shafiq, M. Z., Tabish, S. M., Mirza, F., & Farooq, M. (2009a). *A Framework for Efficient Mining of Structural Information to Detect Zero-Day Malicious Portable Executables*. Pakistan. Retrieved from https://www.researchgate.net/profile/Fauzan_Mirza/publication/242084613_A_Framework_for_Efficient_Mining_of_Structural_Information_to_Detect_Zero-Day_Malicious_Portable_Executables/links/0c96052e191668c3d5000000/A-Framework-for-Efficient-Mining-of-Structu

Shafiq, M. Z., Tabish, S. M., Mirza, F., & Farooq, M. (2009b). PE-miner: Mining structural information to detect malicious executables in realtime. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-04342-0_7

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical*

Journal, *27*(July 1948), 379–423. https://doi.org/10.1145/584091.584093

Shijo, P.V. Salim, A. (2017). *Integrated Static and dynamic analysis for malware detection.* Indian Institute of Technology Kanpur. Retrieved from https://security.cse.iitk.ac.in/sites/default/files/15111029.pdf#page33

Siddiqui, M. (2008). *Data Mining Methods for Malware Detection*. University of Central Florida.

Sorokin, I. (2011). Comparing files using structural entropy. *Journal in Computer Virology*, *7*(4), 259–265. Retrieved from http://www.csee.umbc.edu/courses/undergraduate/CMSC491malware/umbconly/Sorokin2011notesCKN.pdf

Stallings, W. (2017). *Cryptography and Network Security Principles and Practice* (7th ed.). Hoboken, New Jersey: Pearson Education.

Su, J., Vargas, D. V., Prasad, S., Sgandurra, D., Feng, Y., & Sakurai, K. (2018). Lightweight Classification of IoT Malware based on Image Recognition. Retrieved from http://arxiv.org/abs/1802.03714

Sun, L., Versteeg, S., Boztas, S., & Yann, T. (2010). Pattern Recognition Techniques for the Classification of Malware Packers. *Australasian Conference on Information Security and Privacy*. https://doi.org/10.1007/3-540-45450-0

Tabish, S. M., Shafiq, M. Z., & Farooq, M. (2009). Malware detection using statistical analysis of byte-level file content. *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*. Paris, France: ACM. https://doi.org/10.1145/1599272.1599278

Tensorflow. (n.d.). Retrieved February 17, 2018, from http://www.tensorflow.org

Terraform.io. (n.d.). Retrieved January 6, 2018, from https://www.terraform.io/

Tesauro, G. J., Kephart, J. O., & Sorkin, G. B. (1996). Neural networks for computer virus recognition. *IEEE Expert-Intelligent Systems and Their Applications*, *11*(4), 5–6. https://doi.org/10.1109/64.511768

Tran, T. K., & Sato, H. (2017). NLP-based approaches for malware classification from API

sequences. In *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)* (pp. 101–105). https://doi.org/10.1109/IESYS.2017.8233569

Vanderbruggen, T., & Cavazos, J. (2017). Large-scale exploration of feature sets and deep learning models to classify malicious applications. *Proceedings - 2017 Resilience Week, RWS 2017*, 37–43. https://doi.org/10.1109/RWEEK.2017.8088645

VirusTotal. (n.d.-a). Retrieved January 1, 2017, from http://www.virustotal.com

VirusTotal. (n.d.-b). Yara. Retrieved February 3, 2018, from https://github.com/VirusTotal/yara

Wang, Q., Guo, W., Zhang, K., Ororbia, A. G., Xing, X., Giles, C. L., & Liu, X. (2016). Adversary Resistant Deep Neural Networks with an Application to Malware Detection. https://doi.org/10.1145/3097983.3098158

Wojnowicz, M., Chisholm, G., Wallace, B., Wolff, M., Zhao, X., & Luan, J. (2017). SUSPEND: Determining software suspiciousness by non-stationary time series modeling of entropy signals. *Expert Systems with Applications*, *71*, 301–318. https://doi.org/10.1016/j.eswa.2016.11.027

Wojnowicz, M., Chisholm, G., & Wolff, M. (2016). Suspiciously Structured Entropy : Wavelet Decomposition of Software Entropy Reveals Symptoms of Malware in the Energy Spectrum. *Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference*, 294–298. Retrieved from http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS16/paper/view/12978

Wojnowicz, M., Chisholm, G., Wolff, M., & Zhao, X. (2016). Wavelet decomposition of software entropy reveals symptoms of malicious code. *Journal of Innovation in Digital Ecosystems*, *3*(2), 130–140. https://doi.org/10.1016/j.jides.2016.10.009

Wolff, M., & Chisholm, G. (2015). Structural Entropy Analysis for Automated Malware Classification. RSA Security Conference. https://doi.org/r

Wright, J. L. (2009). Neural Network Approach to Locating Cryptography in Object Code, 1–4.

Wright, J. L., & Manic, M. (2010). Neural network architecture selection analysis with application to cryptography location. *Neural Networks (IJCNN), The 2010 International Joint Conference On*, 1–6. https://doi.org/10.1109/IJCNN.2010.5596315

Xiaofeng, L., Xiao, Z., Fangshuo, J., Shengwei, Y., & Jing, S. (2018). ASSCA: API based Sequence and Statistics features Combined malware detection Architecture. *Procedia Computer Science*, *129*(October), 248–256. https://doi.org/10.1016/j.procs.2018.03.072

Xu, L. (2016). *Android Malware Classification Using Parallelized Machine Learning Methods*. University of Delaware.

Xu, L., Cavazos, J., Jayasena, N., & Cavazos, J. (2016). HADM : Hybrid Analysis for Detection of Malware. *SAI Intelligent Systems Conference*, 1037–1047.

Xu, Z., Wen, C., Qin, S., & Ming, Z. (2017). Effective Malware Detection Based on Behaviour and Data Features. In *Smart Computing and Communication* (Vol. 10135). Shenzhen, China. https://doi.org/10.1007/978-3-319-52015-5

Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y., & Sakuma, J. (2017). Malware Analysis of Imaged Binary Samples by Convolutional Neural Network with Attention Mechanism. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security - AISec '17* (pp. 55–56). New York, New York, USA: ACM Press. https://doi.org/10.1145/3128572.3140457

Yan, J., Qi, Y., & Rao, Q. (2018). Detecting Malware with an Ensemble Method Based on Deep Neural Network. *Security and Communication Networks*, *2018*.

Yousefi-Azar, M., Hamey, L., Varadharajan, V., & Chen, S. (2018). Malytics: A Malware Detection Scheme. Retrieved from http://arxiv.org/abs/1803.03465

Yue, S. (2017). Imbalanced Malware Images Classification: a CNN based Approach, 3–7. Retrieved from http://arxiv.org/abs/1708.08042

Yuxin, D., & Siyi, Z. (2017). Malware detection based on deep learning algorithm. *Neural Computing and Applications*, *1*, 1–12. https://doi.org/10.1007/s00521-017-3077-6

Zhao, B., Han, J., & Meng, X. (2017). A malware detection system based on intermediate language. *2017 4th International Conference on Systems and Informatics (ICSAI)*, (Icsai), 824–830. https://doi.org/10.1109/ICSAI.2017.8248399

APPENDICES

APPENDIX A: A TOUR OF MALGAZER'S SOURCE CODE REPOSITORIES

Introduction

This appendix will present a tour of the Malgazer source code and repositories. First, a quick tour of the Malgazer source code will be presented, since most of the source code has been explained previously. Next, the Terraform files that were used to compute the experiments at scale will be presented. After that, the training data and trained classifiers will be presented. Because of the large size, the training data and trained classifiers were stored at a different repository, but everything is linked through the Malgazer source code repository, so users can easily locate the data. Next, the Docker files for the web application are presented before concluding with a tour of the web application source. Please note that this appendix was written in late 2018 and will coincide with the "dissertation" git branch. Improvements to the repositories are likely, so users interested in the current status of this research project should read the readme files in the git "master" branches.

Malgazer's Python Libraries

The Malgazer Python libraries are found in the main source code repository (Jones, 2017b) under the "library" directory. There are several library files in this directory: entropy.py, files.py, ml.py, plots.py, utils.py. These modules were previously discussed in the research methodology chapter, so they will not be reiterated here. These libraries were used heavily by other Malgazer scripts found in the root directory of the repository such as "extract_gist_features.py", "extract_rwe_features.py", "train_classifier.py", etc. Each of these scripts served a purpose and will be mentioned below:

- **extract_rwe_features.py**: This script calculated the RWE features from directories of malware. The user is able to specify the running window entropy window size and resampled data size. This script utilized the "Utils" library discussed previously. The RWE data is saved in HDF files, which is a common file type in data science.
- **extract_gist_features.py**: This script calculated the GIST features using the same methodology as the RWE features script above.
- **vt_intelligence_hunting_downloader.py**: This script downloaded VirusTotal information for classifications.
- **build_classifications_from_vt_data.py**: This script translated the raw VirusTotal data into Malgazer classifications. This script was executed on the data returned from the previous script, above.
- **train_classifier.py:** This script trained the classifiers. This was the script that was executed for over 200 experiments in this dissertation.

Terraform Files

The Terraform files that were used to scale the computations can be found in (Jones, 2018). While this dissertation will not go in depth about the usage of Terraform, the portions of Terraform that are specific to Malgazer will be presented here. Inside the Malgazer Terraform repository, there are two directories. One directory is "malgazer-training-aws" and the other is "malgazer-training-azure". The first directory built training resources in Amazon AWS, while the second directory built a very similar cloud resource in Microsoft Azure. AWS was used for the neural network training while Azure was used to train the traditional machine learning algorithms. Each will be discussed in their own sections below.

Amazon AWS

This Terraform code was used to create AWS resources as needed to train the classifiers. In 2018, AWS offered spot instances, which is a more cost-effective means of purchasing cloud resources. Spot instances made it possible to train neural networks with GPUs without the upfront costs of purchasing a GPU. It was possible to rent a GPU for a few hours of

computation. There are several files in this directory, and each file will be briefly discussed below:

- **scripts**: This directory contains several useful bash scripts to setup cloud training resources. Once the resource is up, one can execute "/tmp/build_training.sh" to install all the required dependencies on the training computer. If the cloud resource had a GPU installed, the script "build_gpu.sh" installed the required dependencies for a GPU. Lastly, running the "get_training.sh" script expanded the training data sets once they were transferred to the instance.
- **backends.tf**: This file configures where the Terraform state file will be saved.
- **keys.tf**: This file configures the SSH key information. The user must create their own SSH key and name it "mykey.cert" and "mykey.cert.pub" in this directory.
- **networking.tf**: This file configures the spot instance's networking information.
- **outputs.tf**: This file lists the outputs that will be generated when Terraform is executed. This file will list the IP address of the cloud resource after it is created so that the user can connect via SSH.
- **providers.tf**: This file configures Terraform for Amazon AWS.
- **spot-requests.tf**: This file describes the spot requests.
- **user-data.txt**: This file executes a management script on the cloud resource that is specific to spot requests and hibernation.
- **variables.tf**: This file defines all of the variables as user must supply to instantiate this resource.

In order to use the Terraform source code, one must create a "terraform.tfvars" file that instantiates all of the variables listed in the "variables.tf" file. Without this file, Terraform will not work. In addition, the SSH key needs to be created with a command line SSH utility and saved as "mykey.cert" and "mykey.cert.pub". Once these steps have been completed, one could create the cloud resources with "terraform apply" or destroy them with "terraform destroy".

Microsoft Azure

The concepts for Terraform with AWS is very similar to Azure, with some slight differences. In 2018, Azure did not offer a comparable service to "spot instances", so that

concept will not apply. Most other concepts will transfer, but the names are slightly different. Instead of "instance", Azure denotes computers as "virtual machines". The following files are in the Azure portion of Malgazer's Terraform repository:

- **scripts**: This directory contains several useful bash scripts to setup cloud training resources. Once the resource is up, one can execute "/tmp/build_training.sh" to install all the required dependencies on the training computer. Running the "get_training.sh" script expanded the training data sets after they were transferred to the instance.
- **backends.tf**: This file configures where the Terraform state file will be saved.
- **networking.tf**: This file configures the virtual machine's networking configuration.
- **outputs.tf**: This file lists the outputs that will be generated when Terraform is executed. The endpoint information for the cloud resource will be returned, and the user can use SSH to connect to it.
- **providers.tf**: This file configures this repository for Microsoft Azure.
- **resource_groups.tf**: This file configures the groups for Azure cloud resources.
- **variables.tf**: This file defines all of the variables a user must supply to instantiate this resource.
- **virtual_machine_1.tf**: This file configures the virtual machines information.

Again, in order to use the Terraform source code, one must create a "terraform.tfvars" file that instantiates all of the variables listed in the "variables.tf" file. Without this file, Terraform will not work. Once these steps have been completed, one could create the cloud resources with "terraform apply" or destroy them with "terraform destroy".

Training Data

The training data was too large to store on GitHub, so it was saved to a DropBox account and linked from the GitHub readme files. Please note that this dissertation was written in 2018, so the data could have been moved to other file sharing services as appropriate. The readme file

in the “master” git branch of the Malgazer source code will contain any of the details that may have changed since this dissertation was printed, along with the link to the training data sets.

The training data consists of two main directories: “GIST” and “RWE”. Each directory stores the data for its respective feature type. While the GIST data set was relatively simple to store, several RWE files were stored for different running window size and resampled running window entropy series length. In addition, these directories store the classification information and SHA256 hashes of every malware sample used throughout this research.

Trained Classifiers

The trained classifiers were also too large to store directly to GitHub, so the classifiers were saved to a DropBox account, just like the training data discussed above. The most current link is available in the readme file of the “master” git branch of main Malgazer source code repository. The trained classifiers can be located in the “training\final_training\classifiers” subdirectory.

Inside this directory there are subdirectories used to categorize the classifiers by input feature type and machine learning algorithm. The GIST and Malgazer classifiers were created and stored as “ml.dill” Python dill pickle files. The “ml.dill” classifiers can be used with the Python web application, which will be discussed shortly. The pickle files can also be loaded with the “dill” library in Python, directly, for users wishing to do so.

Docker Files

To create the Malgazer web application, Docker is required. There are two Dockerfile configuration files in the “docker” directory of the Malgazer source code repository. One is called “Dockerfile-base” and the other is called “Dockerfile-nginx”. The first file creates the images for the API and web portions of the Malgazer web application. The second Docker file creates the image for the load balancer for the Malgazer web application. These files are built by another file in the root directory named “docker-compose.yml”.

For a user to create the Malgazer web application, Docker must first be installed, then the user can create all of the required images with the “docker-compose build” command. After the images are built, the user can type “docker-compose up” and the web application will be created.

Then, the user can browse the URL https://localhost to use Malgazer. Most users will need to modify the Docker files very little, if any.

Malgazer Web Application

The last component discussed in this chapter is the Malgazer web application source code. When the web application is started with the "docker-compose up" command, the classifier named "ml.dill" in the "classifier" directory is loaded into the Malgazer web application. Any malware samples uploaded to the web application will be saved in the "samples" directory by its SHA256 hash.

The Malgazer web application source code can be found in the "docker" directory (Jones, 2017b). There are several subdirectories below the "docker" directory, and each will be briefly explained below.

The "api" Directory

This directory holds the API script. The API is the logic of the web application, as even the web application communicates with the API for work to be completed. Most configuration options are set via environment variables within Docker and would not need to be modified here for most users.

The "common" Directory

This directory contains logic that is useful for multi-user mode. The process of emailing and creating user tokens is stored here. Users running the web application in single user mode will not be interested in this directory.

The "db_models" Directory

This directory contains the database models. There are several tables in a Postgresql database the web application and API use, so this file declares the structure of the tables. The database data can be used as objects and the database communications are abstracted through the Python SQLAlchemy module. Interested users can find the table structure for the database here.

The "files" Directory

This directory contains the configuration files and scripts for the various components to the Malgazer web application. The files in this directory are copied to the Docker containers when they are created, and they contain entry point scripts or other configuration files useful to the running system.

The "logs" Directory

This is the directory where logs are kept when the Malgazer web application is running. The logs stored here are plain text.

The "scripts" Directory

This directory contains useful scripts. The scripts in this directory can assist with the creation of a Docker registry, if needed, for a Docker swarm configuration.

The "web" Directory

This directory contains the web application logic. This directory also contains the templates and other visual components for the website. Users interested in changing any of the web application's visual appearance or business logic can find the relevant files in this directory. The web application was developed with Python Flask, and users that venture into this directory may wish to be versed in Flask before modifying the logic.

Summary

This appendix provided a tour of the Malgazer repositories. There were three main repositories explained in this appendix. The first repository was the main Malgazer source code repository on GitHub (Jones, 2017b). All information for Malgazer can be found from this repository in the "master" branch. If a user wishes to see the state of the source code when this dissertation was published, the "dissertation" git branch will contain the matching information.

The second repository was a GitHub repository that contained the Malgazer Terraform cloud infrastructure code (Jones, 2018). This repository was used to create cloud resources on

demand for classifier training purposes. There are a number of important steps the user must follow to use the terraform files, and those steps were presented in this appendix.

Lastly, the training data and trained classifiers are available via a DropBox link that is listed in the Malgazer source code repository readme. The files were many gigabytes in size and could not be stored with the source code in GitHub, so they were made available via the link listed in the readme file.